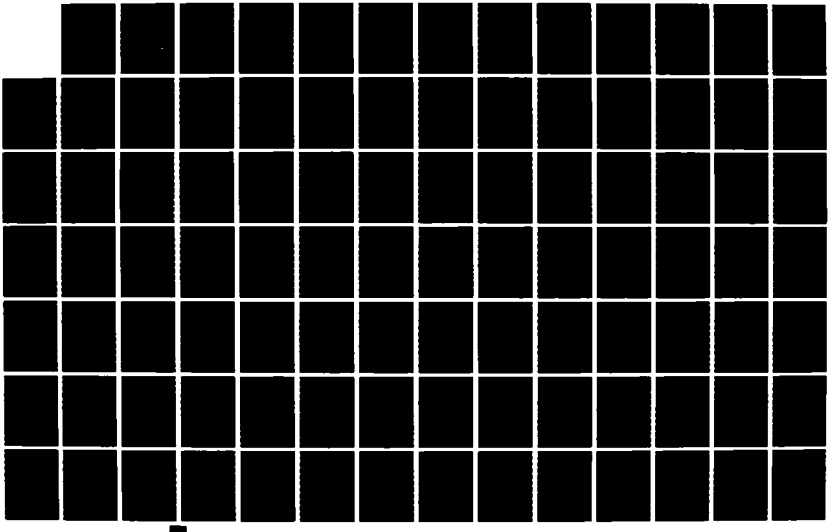
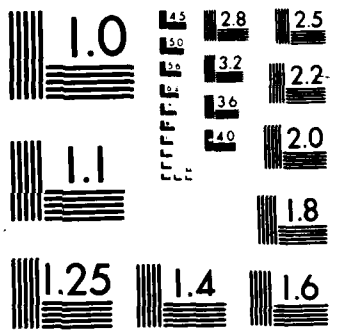
AD-A172 502 CONTROLLING INFERENCE(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE D E SMITH APR 86 STAN-CS-86-1107 1/3
N00014-81-K-0004

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

April 1986

Report No. STAN-CS-86-1107

Controlling Inference

by

David E. Smith

Department of Computer Science

Stanford University
Stanford, CA 94305

April 9, 1986

Controlling Inference

by

David E. Smith

This report is a slightly revised version of a dissertation submitted to the Department of Computer Science of Stanford University on August 15, 1985, in partial fullfillment of the requirements for the degree of Doctor of Philosophy

Department of Computer Science
Stanford University
Stanford, CA 94305

Acknowledgements

Throughout my apprenticeship at Stanford I have accumulated many personal and intellectual debts. By far the largest debt is to my principle advisor, Mike Genesereth. It is difficult to overstate the impact that Mike has had on both the content and presentation of this work. Those who know Mike will undoubtedly see the effects of his criticism and guidance throughout this document.

Quite apart from his direct influence on this work, I have many less tangible things to thank Mike for. I have always enjoyed a very close working relationship with Mike and have fond memories of our many marathon research meetings. Sometimes we both sat, silent, staring at the same blackboard, neither of us feeling compelled to interrupt the others train of thought. In this respect, Mike has treated me much more like a friend and colleague than a student. Yet, by his own example and by his incisive comments, Mike has taught me a great deal about separating issues, about organizing material for written and oral presentation, about giving good constructive criticism to others, and about good scholarship. These are the lessons that I value most.

Special thanks to the other three members of my reading committee, Bruce Buchanan, Pat Hayes and Ted Shortliffe. There is a deliberate tension in my committee between the theoretical and applied. I asked Bruce and Ted to serve, partly to keep me honest, and partly to benefit from their vast knowledge of practical systems and examples. Pat, on the other hand, has always served as a source of inspiration for me. In an invited lecture at IJCAI-81, Pat encouraged a concentration on "content" rather than "form". His "Ontology for Liquids" paper [Hay84a] is a model in this regard and I have, in some respects, attempted to copy that style.

Despite the fact that this work is not the primary research interest of any of these three, all three graciously gave their time to read, consider, critique and offer different perspectives on this work. All of my committee members have probably allowed me more leash than I deserve. They have patiently looked on while I explored blind alleys, and through this patience have allowed me to discover my own research and methodological niche. I only regret that I have not made greater use of these formidable resources.

Many others have contributed directly or indirectly to the research described

here. Thanks to

- Bruce Buchanan, Ed Feigenbaum, Doug Lenat, Penny Nii and Mark Stefik for taking me under their wings during my first two years at Stanford and for involving me in research.

- Mark Stefik, Doug Lenat and Jon Doyle for stimulating and fostering my interest in representation issues and issues of intelligent system architecture. My own understanding of both these areas was, I believe, a necessary prerequisite to the work described here.

- Drew McDermott for detailed comments on much of the technical material in Chapter 5.

- Matt Ginsberg for assistance with some of the derivations in Chapters 5 and 6, and for finding counterexamples to some of my conjectures. Matt has proven his ability to do probability calculations at light speed.

- Tom Dietterich, Jeff Finger, Glenn Kramer, John Kunz, Narinder Singh, Vineet Singh and Richard Treitel for using fragments of my work in their own research. Without intending, this group has continually tugged and pulled at my ideas and has provided me with many crisp and interesting examples.

- Jim Bennett, Jeff Finger, Russ Greiner and Jock Mackinlay for the many "fireside" discussions of AI that have kept me excited about the field.

- Ed Feigenbaum, who doesn't realize what he has done for me. Ed has always served as my "ghost of Christmas future". Whenever I begin to get caught up in some useless, esoteric problem, Ed appears in my dreams, wailing and clanking chains, and demanding that I justify my research.

- the members of the Logic Group at Stanford for being a receptive and sympathetic audience (and for all the good cookies).

- Jim Bennett, Jan Clayton, Tom Dietterich, Russ Greiner, Jock Mackinlay and Vineet Singh for editorial assistance with text and pictures.

- Jan Clayton for typing assistance, and for resisting the urge to throw me out of the house with the cat.

This work was supported by ONR contract N00014-81-K-0004.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Importance of Control

There is little disagreement about the need for controlling search in problem solving. In a recent article on the current state of Artificial Intelligence (AI) Lenat writes [Len84]:

> *Most interesting problems ... share the characteristic that they are too complex to be solved by random search, because the number of choices increases exponentially as one proceeds from the first ... decision point.*
>
> *Therein ... lies the essence of intelligence: finding ways to solve otherwise intractable problems by limiting the search for solutions.*

The necessity of controlling search is the single most important maxim for building intelligent systems; random or exhaustive search is not computationally feasible for most interesting problems.

Inference problems are no exception. Consider the following simple collection of rules about various kinship relations:

$$\begin{aligned}
&Sibling(c,d) \Longleftrightarrow Sibling(d,c)\\
&Brother(c,d) \Longleftrightarrow Sibling(c,d) \wedge Male(d)\\
&Sister(c,d) \Longleftrightarrow Sibling(c,d) \wedge Female(d)\\
&Sibling(c,d) \Longleftrightarrow \exists p Parent(c,p) \wedge Parent(d,p) \wedge c \neq d\\
&Father(c,p) \Longleftrightarrow Parent(c,p) \wedge Male(p)\\
&Mother(c,p) \Longleftrightarrow Parent(c,p) \wedge Female(p)\\
&Child(p,c) \Longleftrightarrow Parent(c,p)\\
&Son(p,c) \Longleftrightarrow Child(p,c) \wedge Male(c)\\
&Daughter(p,c) \Longleftrightarrow Child(p,c) \wedge Female(c)
\end{aligned} \tag{1.1}$$

These axioms state that the sibling relation is symmetric, brothers are male siblings, sisters are female siblings, siblings share a common parent, mothers are female

parents, fathers are male parents, child and parent are inverse relationships, sons are male children and daughters are female children, respectively.

Given a small collection of kinship data, such as

$$\begin{array}{l} Brother(Rob, Larry) \\ Father(Rob, Bill) \\ Mother(Rob, Terry) \\ Father(Pat, Bill) \\ Father(Larry, Bill) \end{array} \tag{1.2}$$

and a question to answer such as "find all of Rob's siblings", the number of different possible inference steps is immense. Consider only the backward inference steps for a moment. There are four different axioms that can initially be used to reduce the initial problem to a subproblem (the symmetry axiom can be applied in two different ways). Suppose that we choose to apply the fourth axiom, which says that two individuals sharing a common parent are siblings. There are three different possible conjuncts to explore for this subproblem. For each of the first two conjuncts there are four different axioms that apply. Discounting the fact that it is possible to apply each of the if-and-only-if axioms over and over again, there are still more than forty different backward inference steps possible in the solution of this apparently simple problem. The number of possible forward inference steps is even larger, since it includes steps that do not lead to the goal and steps that combine rules to make new rules. We did not even include axioms describing cousins, aunts, uncles, grandparents, in-laws, step-relations, or ancestry. As Pat Hayes has pointed out [Hay84b], the everyday, common-sense world is rich in such interconnected concepts. Thus, while we might regard this example as a "toy" problem, it is a difficult and all too common toy. The lesson from this toy problem is just as relevant to complex practical problems; random exhaustive inference is not feasible because the number of potential inference steps and conclusions is combinatorially explosive in the number of facts available. Inference must be controlled.

1.2 The Dimensions of Control

Many different techniques have been used for controlling inference in AI programs. These techniques vary along many different dimensions. At a fundamental level, the adequacy or performance of a control method is measured by two different criteria:

1. *Completeness*

2. *Cost*

The first criteria, completeness, means the extent to which a control procedure will sanction the inference steps necessary to find solutions to the problems of interest. This is different from the notion of *logical completeness*, the extent to which a control procedure can recommend any logically valid inference step. The effectiveness of a control procedure is measured by the former concept rather than the latter, because we only care about those inference steps that are necessary for solving the range of problems that a system is expected to solve.[1]

The second criteria, cost, refers to how expensive it is to solve the problems of interest using the control method. There are two different components to this cost:

1. *Inference Cost*
2. *Control Cost*

Inference cost is the cost of actually performing the steps in the inference strategy recommended by a control method. This is a measure of how good the control advice is. Control cost is the cost of making the necessary control decisions for the problems of interest.

The three criteria – completeness, low inference cost, and low control cost – are, in some sense, at odds with each other. It is relatively easy to imagine control procedures that satisfy any two of the three criteria. For example, if we totally sacrifice completeness we could simply build an overzealous control procedure that throws out all inference steps. The problem solver would answer "I don't know" to every query, but it would do so *very* fast. If we chose to ignore inference cost instead, a dumb strategy like breadth-first resolution would satisfy the other two criteria. Alternatively, by completely ignoring control cost, we could build an idealistic control procedure that first solved each problem by every possible approach, and then directed the inference engine to follow the ideal sequence of inference steps.

All three of the above possibilities are a bit silly, but they serve to point out the importance of the three basic criteria in the choice of a control method. As control cost goes up, inference cost can be reduced. As control cost goes down, inference cost goes up, or completeness is seriously sacrificed.

In addition to these three basic criteria, control methods can be seen to vary along several other dimensions

1. *Domain-dependence* – the degree to which control principles depend upon the problem solving domain.
2. *Locality* – the degree to which control decisions rely on local information about the inference steps under consideration, as opposed to global information about the database.

[1] For some systems this may entail logical completeness. But logical completeness is not required for many more specialized problem solving tasks.

3. *Control Complexity* – the computational complexity involved in making control decisions.[2]

4. *Control Separation* – the time at which control decisions are made, relative to the time when they take effect.

The first two factors affect the performance of a control procedure in subtle ways. The last two criteria directly govern the control cost of a particular procedure.

1.2.1 Domain-dependence

Control methods can vary from being highly domain-dependent to completely domain-independent. For example, the control principle

Prefer rules with the predicate ImmunoSuppressed in their premises

is highly dependent on the domain of medicine, since it applies only to rules in that domain.[3] In contrast, the control principle

Prefer rules with only a single conjunct in their premise

is completely domain-independent because it applies to rules independent of their subject domain.

Many well known control strategies are completely domain-independent. The unit-preference strategy, the linear-input strategy and depth-first backward-chaining all fall into this category. I will sometimes refer to methods that are completely domain-independent as *weak* or *syntactic* control principles.

There is considerable middle ground between the two extremes of highly domain-dependent control and completely domain-independent control. For example, a principle like

Prefer rules that deal with malfunctions that are highly likely

applies to diagnostic problems in general, as opposed to problems in just one particular diagnostic domain like medicine. Yet, the use of this principle requires domain-dependent information about likely malfunctions. As another example, a principle like

[2]Note that this is not the only factor governing the net control cost. *Control Separation* and the frequency with which control control decisions are used are also factors.

[3]The reader's intuitive notions about the meanings of domain-dependence and domain-independence are probably adequate for present purposes. It is actually quite tricky to provide a precise definition of these concepts. For the compulsive theorist, Appendix A contains formal definitions of these concepts.

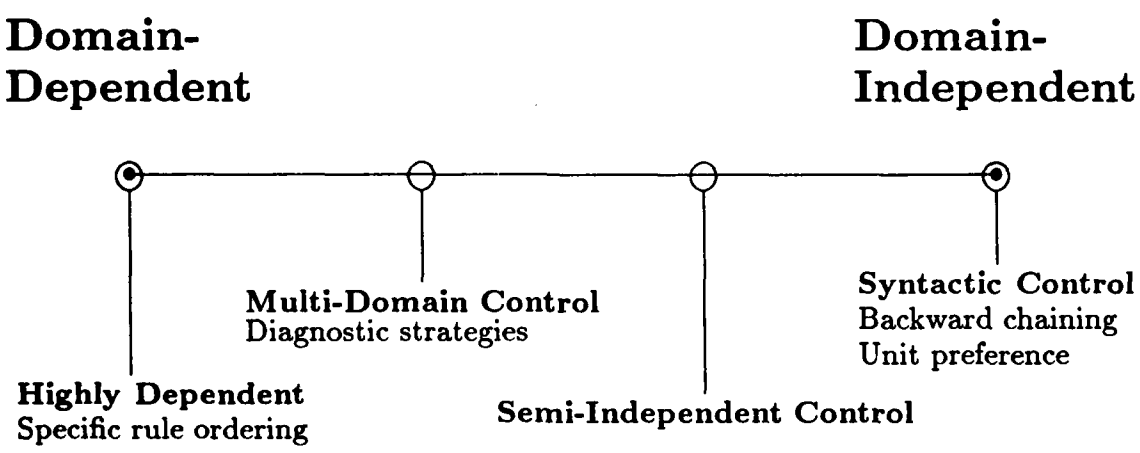


Figure 1.1: The domain-dependence spectrum

Prefer the conjunct that will have the fewest solutions

is, by itself, domain-independent. Yet, the use of this principle requires information about the potential number of solutions to different conjuncts. This information is domain-dependent, since it is a function of the contents of the database, which in turn is a function of the domain under consideration. We will refer to this kind of control – domain-independent control principles that require global information about the characteristics of the database – as *semi-independent* control. The different possibilities along the spectrum of domain-dependence are compared in Figure 1.1.

1.2.2 Locality

Control methods can vary in the extent to which they rely on local and global information to make control decisions about inference steps. The unit-preference strategy is strictly a local control method. It relies only on information about the structure of a fact to make control decisions about the corresponding inference steps. A strategy like depth-first backward chaining is slightly less local, since it depends on information about the predecessors of a step. At the opposite end of the spectrum, a principle like

Prefer the conjunct with the fewest solutions

depends on global information about the number of facts of a particular form derivable from the database.

Most control methods used in AI rely on local, rather than global information. In contrast, semi-independent control, like that introduced in the previous section,

involves the use of global information about the characteristics of the database. The locality of a control method is usually a good indicator of the computational complexity of control decisions.

1.2.3 Control Complexity

Control complexity refers to the amount of computation involved in making control decisions. *Shallow* control procedures do little computation in reaching control decisions. In a sense, such control decisions are made *manually* by the user who develops the control procedure (assuming that there is no learning or caching of control decisions). *Deep* control methods do more computation in making control decisions. In a sense, control decisions are derived *automatically* by the system from more fundamental principles. In general, domain-dependent and domain-independent methods can lie anywhere along the computational complexity spectrum. However, most commonly used domain-dependent and domain-independent control techniques are shallow. For example, domain-dependent rule ordering like

Rule 58 is better than rule 69

requires little control computation to effect. Likewise, the unit-preference strategy is shallow, since relatively little computation is involved in deciding whether or not expressions are unit clauses. In contrast, the control principle

Prefer rules dealing with immunosuppression

and

Prefer malfunctions that are highly likely

require considerable computation to realize. Typically, methods that rely on global, rather than local information, tend to require more computational effort. In particular, semi-independent control often requires considerable control computation. Dealing with the potentially expensive computation required in semi-independent control is a matter that is considered in detail in Chapter 7.

The depth of control reasoning has a direct effect on the total cost of control reasoning, but it is not the only factor. Control separation also has an influence on total control cost.

1.2.4 Control Separation

Control separation refers to the matter of *when* control computation takes place relative to when control decisions take effect. If there is little computation involved in making control decisions, this is not an issue. But when control computation

becomes expensive, the question becomes critical. A control decision can be made by a system at any time, from the moment of initial system construction up to the moment that the decision must take effect. We refer to these two extremes as *compile-time* and *run-time* control respectively. As an example, imagine a system that automatically determines the specific order in which to try rules as they are entered into the database. This is compile-time control. In contrast, a system that decides which rule to use, just before actually using the rule, is engaged in run-time control. There are many possibilities in between.

The time at which control decisions are made affects the frequency with which control reasoning must take place. If rules are ordered at compile-time, the ordering need only be done once. In this case the control cost is amortized over all subsequent queries. On the other hand, if ordering decisions are made at run-time, control computation must take place for every inference step. This profoundly affects total control cost. The issue of when to do control is considered in Chapter 7.

1.2.5 Other Dimensions

There are other dimensions of control, in addition to those just discussed.

1. *Explicitness* – the extent to which the control process is implemented as an explicit reasoning process.

2. *Cooperativeness* – the ability of the inference engine to follow arbitrary control advice.

However, these are dimensions of control implementation rather than of control effectiveness. Although control information is represented in an explicit declarative fashion in this dissertation, this is largely incidental to the enterprise. Implementation issues are discussed further in Chapter 8.

1.3 History

> *Adam was the only man who, when he said a good thing, knew that nobody had said it before him.*
>
> – *Mark Twain*

1.3.1 Domain-independent Control

The notion of controlling inference is not a new idea in AI. In fact recognition of the need for control, and the use of simple control, can be found in the earliest

systems. In their description of the Logic Theorist (LT), Newell, Shaw and Simon [NSS63] devote pages to arguments and data supporting the assertion that unguided inference is completely intractable for proving propositional calculus theorems in Russell and Whitehead's *Principia Mathematica*.

> *When sequences of expressions are produced by a simple and cheap generator, the chance that any particular sequence is the desired proof is exceedingly small. This is true even if the generator produces sequences that always satisfy the most complicated and restrictive of the solution conditions: that each is a proof of something. The set of sequences is so large, and the desired proof so rare, that no practical amount of computation suffices to find proofs by means of such an algorithm.*
>
> *If LT is to prove any theorems at all it must employ some devices that alter radically the order in which possible proofs are generated, and the way in which they are tested.*

To overcome this problem, LT employed the strategy of working backward from goal statements rather than forward from given axioms. It also used several simple heuristic criteria for deciding which of the different possible subgoals to work on.

The problem of controlling inference was also manifest in natural deduction and resolution systems throughout the sixties and early seventies. One of the earliest natural deduction systems was constructed by Fischer Black in the early sixties. Black admits [Bla68, page 393],

> *The main trouble with the system is that it is too slow in solving practical problems.*

Black was not just referring to machine speed, but rather to the size of the space of legal inferences that could be made from the available facts and goal statement.

Numerous control measures were invented throughout this period to help guide the inference process, both in resolution and natural deduction systems. Among the better known are the set of support strategy, linear input strategy, ancestry filtration, and unit preference strategy. A good review of these strategies can be found in [Nil80]. What is common to all of these control strategies (including those in the Logic Theorist) is that they are

1. *Domain-independent* – they are not specific to the domain of application.

2. *Local* – they rely only on local information about the available inference steps to make control decisions.

3. *Shallow* – they require relatively little control computation.

4. *Heuristic* – they will not necessarily result in the best strategy, or even a good strategy for solving a problem.

While the domain-independence and simplicity of these control methods is appealing, they are too weak, and too arbitrary to effectively control the inference process in many situations. In his discussion of QA3 [Gre69, page 204], Cordell Green writes,

> *For harder problems it may be necessary to be able to add subject dependent search heuristics or 'advice' for particular problems.*

1.3.2 Domain-dependent Control

Virtually all expert systems built in AI, from early systems like DENDRAL [LBAL80] and MYCIN [BS84] to recent commercial successes like R1 [McD80], have relied on powerful domain-dependent control in addition to syntactic control for guiding search. For example, although MYCIN relies on the syntactic control implicit in the use of depth-first backward chaining, there is also a great deal of domain specific control knowledge built in. Most of it is implicit in the ordering of rules in the system's rule-base, the ordering of conjuncts in rule premises, and in the use of special tricks like screening clauses. There is also domain specific control information recorded explicitly in parameters such as ASKFIRST and ASKLAST.

Recently, the trend in expert systems work is towards making domain-dependent control information more explicit. For example, Davis [Dav80] used production rules to record information about the order in which domain rules in MYCIN should be applied.[4] Like the work of Davis, Clancey [CL81] has been further explicating the control built into MYCIN. However, Clancey has been uncovering more general control principles that apply to what Clancey refers to as *heuristic classification problems* [Cla84].

1.3.3 Semi-independent Control

There has been relatively little development and use of semi-independent control principles in AI. One of the earliest uses was by Gelernter in the Geometry Theorem Proving Machine [Gel63]. Gelernter demonstrated that, in trying to prove a theorem, many of the possible subproblems could be eliminated by searching for a counterexample before actually trying to prove the subproblem. The preview

[4] There is a great deal of work on architectures for supporting explicit representation of control information. For example, [McD77], [dDSS79], [Dav80], [CL81], [Doy80], [GS82], [Lai83] and [Hay85]. However, this is *not* the primary focus of this dissertation.

mechanism used in MYCIN [Sho84] is based on the same principle. In MYCIN, conjunctions are inspected before the backward chaining mechanism is applied to any of the conjuncts. If any of the conjuncts are known to be false, the entire subgoal is discarded.

The thesis work of King [Kin81] can be seen as an extension of the control principle pioneered by Gelernter. King applies forward inference to conjunctive queries to find consequences of the conjunction that will help reduce the amount of search required to find solutions to the conjunction. Such consequences are added to the front of the conjunction. Gelernter's principle can be seen as an extreme case of this, where *false* is derived from a goal expression and is then conjoined on to the goal expression.

Another semi-independent control method was pioneered by Sproull [Spr77]. Sproull assumed the availability of a priori estimates for the probability of success of various actions available to a planner. He then used these estimates to formulate optimal plans. This work is discussed in greater detail in Chapter 6.

Work on semi-independent control has also been done within the database community. Warren [War81] and others (see [Ull82] for an excellent summary) use information about the sizes of sets to order conjunctive database queries. This work is discussed in Chapter 5. Ullman [Ull84] has also been using a form of semi-independent control similar to that used in connection graph theorem provers [Kow75,Sti82] for locating inference paths that have a non-zero probability of success. This work is discussed in Chapter 6.

1.4 The Importance of Domain-Independent Control

As we indicated in the previous section, expert systems have relied on a combination of syntactic and domain-dependent control methods. For example, the MYCIN system [BS84] uses the domain-independent method of depth-first backward chaining together with domain-dependent control provided by ordering of the premise clauses in each rule and ordering of the rules in the system's database.

Unfortunately, not all problem solving tasks submit to domain-dependent control methods. In particular, systems that must handle a large variety of different problems cannot depend on domain-dependent control for their source of power. For example, in building a mobile robot the number of possible goals that could be expected of the robot is enormous, perhaps even infinite. It would not be feasible to characterize and provide domain-dependent control advice for every one of those possible goals.[5] In the case studies of Chapters 3 through 6 we will show

[5] In contrast to the mobile robot, a specialized diagnostic system like MYCIN always has the

other examples of problem solving situations where it is not feasible to provide domain-dependent control information for all possible problems that might occur.

Of course, for applications like a mobile robot (where the range of possible problems is immense), the vast majority of problems may fall into only a few special categories. In other words, ninety-nine percent of the actual problems may constitute only one percent of the set of feasible problems. As a result, even for mobile robots, it may be possible (and advantageous) to handle the common cases by providing domain-dependent control advice. However, if this is the only control available, the robot would be in serious trouble when it encountered a problem not covered by domain-dependent control. If a system must be capable of solving a wide range of problems, no matter how unlikely, or how infrequently they may be encountered, at least some degree of powerful domain-independent control must be available.

There are other reasons for having powerful domain-independent control in an inference system. Even when domain-dependent control is possible for an application, providing all of this information by hand places a tremendous burden on the user. In order to provide adequate control the user must know which problems the system will be expected to solve, what information the problem solver will have available, and must understand the different possible inference paths that the system might follow. All of these are non-trivial requirements. If the user fails to anticipate some goal or situation that the system will encounter, the system may fail to solve those problems in an acceptable time period. Worse yet, if the user provides control information that is incorrect for unanticipated problems, incomplete answers can result, because the system may prune important inference steps.

Even with a thorough understanding of the goals and situations that a system will encounter, recognizing when control is necessary is a difficult task. For example, it is only semi-decidable whether or not a given collection of facts can result in an infinite inference loop. It is NP-hard just to discover whether such a collection of facts can lead to an inference loop at all. In other words, for a complex set of facts, it is difficult to determine what inference control is necessary, short of explicitly trying every possible problem solving situation. Even savvy knowledge engineers can be surprised by unexpected control problems.

In short, complete reliance on domain-dependent control requires omniscience on the part of the system builder. Since few of us have that power, the inevitable result is fragile systems; systems that can fail dramatically on problems not anticipated by the system builder. If a system is to be robust in the face of unanticipated problems

---

same task; to find the diagnosis and therapy for the patient's disorder. Although the inference is not the same in every instance, the number of different inference paths that the system can follow is limited. For such an application like this it is possible to characterize good inference strategies in advance for the different possible situations that the system will encounter.

it must have powerful domain-independent means of control.

1.5 Statement of the Thesis

The principle thesis of this work is that

Semi-independent control

1. *is essential for systems that must deal with a wide range of problems,*
2. *can be practically realized using the basic approach proposed in this dissertation.*

In the previous section we argued that domain-dependent control is not generally feasible for systems that must deal with a wide variety of different problems. Practical examples of this will be shown in the case studies of Chapters 3 through 6. Powerful domain-independent control is needed in these cases, but local domain-independent control is too weak and too arbitrary for the control of these inference tasks. Semi-independent control, that makes use of global information about the contents of the database, provides a solution.

We can think of semi-independent control principles as characterizations of the rationale behind good control decisions. Good control decisions are not arbitrary; there is always a reason why they work. Once these reasons are uncovered and recorded, specific control decisions will follow logically from the domain-independent rationale, and simple facts about the domain.[6] The case studies of Chapters 3 through 6 provide detailed examples of novel semi-independent control principles for different inference control problems. These principles are instances of the more general methodology developed in Chapter 2.

Finally, the matter of practicality (efficiency) is an important one. Global, semi-independent control can require considerable computation. Judicious interleaving of inference with control computation can be used to help reduce the overhead of these expensive computations.

1.6 What This Dissertation is Not

This dissertation is *not* about representation. The concern is with the laws of control, not with the language in which they are expressed. Predicate calculus is

[6] Actually, there may be control decisions where the underlying rationale is so complex that it will never be uncovered. In such cases control decisions would have to be derived by generalization or analogy from direct experience.

used throughout this dissertation to express facts about the world and about the problem solving process. This choice should be regarded as incidental. Any other language with sufficient expressive power would do as well.[7]

This dissertation is *not* about problem-solver architecture. As will be discussed in Chapter 8, some of the control methods developed here do have implications for the structure of a problem solver, but this is a consequence of the analysis, not its aim. In short, the concern is with the *substance*, not the *form* of control.

Finally, this dissertation is *not* an attack on the use of domain-dependent control. One of the most important lessons learned in AI is the importance of domain-dependent control in building high performance expert systems. The kind of global, semi-independent control proposed here is not an alternative to powerful domain-dependent control; it is an adjunct. As I argued in Section 1.4, semi-independent control is essential for the unexpected cases where domain-dependent control has not, or can not be provided.

1.7 Organization of the Document

This dissertation is divided into three parts; general approach, case studies, and efficiency analysis. Chapter 2 is devoted to the general approach for semi-independent control. This approach is based on the use of simple utility theory and averaging assumptions for evaluating different possible strategies.

In Chapters 3 through 6 the application of this general theory to four specific, but very different control problems is explored. The first of these (Chapter 3) is the relatively simple problem of halting inference when all of the solutions to a problem have been found. Information about set sizes is used to compute and maintain

[7]There are some serious disadvantages to the use of predicate calculus. First of all, it is verbose. As Mackinlay and Genesereth have pointed out [MG85] specialized languages such as maps and graphs can sometimes be used to express facts in a very concise fashion. These languages have the property that some of the facts are captured *implicitly* by the language. Unfortunately, no single specialized language is sufficiently expressive for our purposes. For the sake of uniformity we will avoid introducing many different specialized languages.

A second disadvantage of predicate calculus is that it is difficult to read and digest. This is largely due to its lack of locality. The collection of facts containing a given symbol are not all collected together in one place. As a result, deciding whether or not a relationship holds between two symbols requires a scan of the list of facts. Several alternative languages have been proposed that do not suffer from this malady. The most notable are semantic network languages (like *Kl-one*) and (to a certain extent) frame languages, both of which enforce at least some degree of locality. These languages also suggest analogous data structures for implementation. Unfortunately, these languages lack much of the expressive power of even first-order predicate calculus.

Similar sentiments have been eloquently expressed by Hayes [Hay77,Hay84b] and McDermott [McD84].

information about the number of solutions for a problem. This information is then used to halt search when all solutions to the problem have been found.

The second case study (Chapter 4) is that of determining how and when to halt recursive inference. Two different kinds of recursion, repeating inference and divergent inference, are distinguished. Repeating inference, and some common special cases of repeating inference, are halted using powerful syntactic techniques. In contrast, the control of divergent inference depends crucially on information about the relations involved. The methodology is similar to that of using invariant assertions in proofs of program correctness and termination.

The third case study (Chapter 5) is about deciding how to order the clauses of a conjunction for efficient generation of the answers by a generate and test procedure. Again, information about the sizes of sets is used to estimate the cost of solving conjunctions by different orderings. Surprisingly, the simple heuristic of choosing the conjunct with the smallest solution set sometimes proves to be a poor strategy. Several *reduction theorems* are developed that help to restrict the number of different orderings that must be examined to find an optimal ordering.

The final case study (Chapter 6) is about choosing appropriate inference steps for single solution problems where there is no redundancy in the search space. Estimates of the probability of success and expected cost for each inference step are used to help compute the utility for different possible strategies. The optimal strategy can then be found using this information. Two reduction theorems are presented that help to limit the number of different possible strategies that must be considered in locating the optimal strategy.

Taken together, the four case studies provide examples of how the general approach of Chapter 2 can be applied to specific control problems.

The primary concern in the case studies is with elucidating the information and reasoning required to make control decisions, not with the efficiency of the control methods themselves. In Chapter 7, the efficiency issue is considered; when it is valuable to expend energy on control, and how much energy should be spent. The interleaving of control and inference is introduced as a way out of the apparent dilemma between time spent doing inference and time spent controlling inference.

1.8 On Reading this Document

Although the background and motivations of readers will vary, a few words of advice are in order when reading this document. I have attempted to make this document accessible to those less formally inclined. For the most part, the document should still be readable if all of the axioms and proofs are skipped.

The casual reader may wish to skim through the details of the case studies (Chapters 3 through 6) on first reading. These case studies are quite detailed and

a thorough understanding of the details is not necessary in order to understand either subsequent case studies or the material of Chapter 7. When the need for specific control information arises, the material in these chapters can be digested more thoroughly. This warning does not apply to researchers interested in search analysis.

Finally, it is my hope that an AI practitioner, concerned with the immediate problems of constructing an expert system, will find the case studies of Chapters 3 through 6 to be a useful resource. Even though the cases are far from exhaustive, in each instance I have attempted to analyze the different control possibilities and indicate both their advantages and weaknesses.

Chapter 2

Approach

2.1 Notation

2.1.1 Facts

As mentioned in Chapter 1, predicate calculus is used throughout this dissertation for writing both base-level propositions (about the world), and meta-level propositions (about problems and the problem solving process). Any other language with sufficient expressive power would do as well.

Traditional infix syntax is used. The standard propositional connectives $\wedge, \vee, \Longrightarrow, \Longleftrightarrow$, and $\neg$ correspond to conjunction, disjunction, implication, equivalence, and negation respectively. Likewise, $\forall$ and $\exists$ correspond to universal and existential quantification. The existential quantifier is extended by letting $\exists_n$ mean "there exists exactly n" and $\exists_{\geq n}$ mean "there exists at least n". Thus, $\exists$ is an abbreviation for $\exists_{\geq 1}$. All constant, function and relation symbols are capitalized, while lower case letters are used for variables. All free variables in facts are understood to be universally quantified with the entire expression as scope.

Braces are used to denote sets (e.g. $\{1,3,5\}$) and angle brackets are used to denote ordered tuples of objects, (e.g. $\langle 1,3,5 \rangle$). The notation $s|t$ denotes the concatenation of two sequences (i.e. "append"). Expressions like $\{y : Loves(x,y)\}$ refer to the set of things that satisfy the given predicate or proposition. This is used as a shorthand for writing an axiom like

$$y \in Lovers(x) \Longleftrightarrow Loves(x,y)$$

and using the functional expression $Lovers(x)$ in place of the above abbreviation. The notation $\|S\|$ refers to the *cardinality*, or number of elements in the set S.

When it is necessary to refer to a base-level expression, it is enclosed in quotation marks, e.g. $Provable(\text{"}Father(A,B)\text{"})$. Lower case Greek letters occurring within quotation marks are meta-variables, i.e. they range over expressions

in the base-level language. From *Provable*("$Father(\alpha, \beta)$"), it is legal to infer *Provable*("$Father(A, B)$").

Finally, the notation $s|_v$ refers to the proposition formed by substituting each of the variable bindings in the set v into the expression s. Thus, $R(x, y)|_{\{x:A\}}$ is the proposition $R(A, y)$. This usage is analogous to the model theory notation for assignment. If a variable occurs in the set v without an explicit binding, the binding is assumed to be a unique skolem constant.

2.1.2 Queries

Queries will be formally represented by three components:

1. *Goal expression* – any arbitrary propositional expression containing zero or more variables.
2. *Output variables* – the variables of interest. This can be any subset (possibly empty) of the free variables in the goal expression.
3. *Number of solutions* requested – either a positive integer, or "all" if all solutions are desired.

Queries are expressed in the form

$$find\ n\ v\text{:}\ p$$

where n, v and p are the number of solutions desired, the output variables and the goal expression, respectively. When the set of output variables is the same as the set of free variables in the goal expression the set of output variables is often omitted. If the set of desired variables is a strict subset of the free variables in the goal expression the other free variables in the goal expression are understood to be existentially quantified. For example, the query

$$find\ \ x\text{:}\ Loves(x, y)$$

is understood to be the same as

$$find\ \ x\text{:}\ \exists y Loves(x, y)\ .$$

This is precisely opposite of the convention for free variables in facts, but is in keeping with the normal duality between facts and goals.

2.2 Definitions

So far terms like *inference*, *control*, and *strategy* have been used rather loosely. I have relied on the readers intuition about what they actually mean. Before proceeding, more precise definitions are needed.

2.2.1 Inference

An inference problem is characterized by a query to be answered from a database of facts using some collection of rules of inference. Only *inference actions* are permitted in finding solutions to the query. This includes two different kinds of logical operations:

1. *Forward deduction* – the derivation of new propositions from known propositions according to some given set of rules of inference (classical or otherwise).

2. *Backward deduction* – the generation of new goals (subgoals) from known goals and known propositions according to a set of rules of inference.

We will often refer to the space of all possible inference steps for a problem. This includes not just those steps that can be performed immediately, but the steps that become possible after these initial steps are performed and so forth. Thus, there is an a priori partial ordering between steps and their successors in the space.

To make this concrete, consider the simple set of axioms

$$\begin{array}{l} P(A) \\ P(B) \\ P(x) \Longrightarrow Q(x) \end{array} \tag{2.1}$$

and the problem of determining whether the goal $Q(A)$ is true. For this problem, the inference space contains a total of five possible inference steps:

1. deriving $Q(A)$ from the facts $P(A)$ and $P(x) \Longrightarrow Q(x)$

2. deriving $Q(B)$ from the facts $P(B)$ and $P(x) \Longrightarrow Q(x)$

3. deriving the subgoal $P(A)$ from the goal $Q(A)$ and the fact $P(x) \Longrightarrow Q(x)$

4. deriving the empty subgoal from the goal $Q(A)$ and the derived fact $Q(A)$

5. deriving the empty subgoal from the derived subgoal $P(A)$ and the fact $P(A)$

The first two are forward inference steps and the remaining three are backward inference steps. Step 4 requires that step 1 be performed first and step 5 requires that step 3 be performed first. The space of possible inference steps and the partial ordering among them is summarized by the graph in Figure 2.1.

An inference step is said to be *superfluous* if it does not lead to a solution to the problem. Step 2 is superfluous in the example above. A set of inference steps, s, is said to be *redundant* with another set of steps s' if every solution that can be produced by the set s can also be produced by the set s'. In the example above, steps 1 and 4 are redundant with steps 3 and 5, and vice versa.

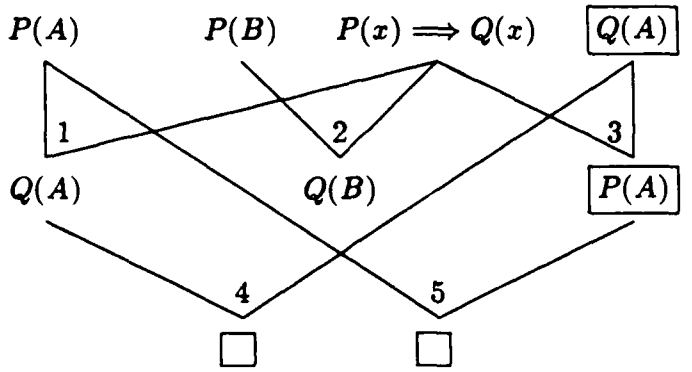


Figure 2.1: Graph of possible inference steps and the partial ordering among them. Goals have boxes around them to distinguish them from facts. The notation is similar to that used in resolution refutation graphs; a bundle of arcs from a group of facts and goals to a fact or goal lower in the graph means that the higher facts and goals are used to derive the lower fact or goal.

An inference problem is said to be solved, if the desired number of solutions has been found to the query. A solution to a query is a set of bindings for the output variables such that when the bindings are substituted into the goal expression, the goal expression is provable from the database.

2.2.2 Strategies

Following Simon and Kadane, [SK75] any legal sequence of inference steps is referred to as a *total strategy* or just *strategy* for short.[1] More precisely, a total strategy is a totally ordered subset of the set of all possible inference steps such that the predecessors of each inference step precede it in the sequence. Thus, the predecessors of each step in the sequence must also be in the sequence, and the partial ordering between steps and their predecessors must be respected. In the example shown in Figure 2.1, the sequences $\langle 1,3\rangle$ and $\langle 1,3,5,4,2\rangle$ are both legal strategies, while $\langle 5,3\rangle$ and $\langle 2,4,1,3\rangle$ are not because they violate the partial ordering between steps 1 and 4 and steps 3 and 5. For this example, there are 90 different legal strategies that can be formed from the five different inference steps.

[1] A *partial strategy* could be defined as arbitrary constraints on the sets of inference steps allowed at each stage of the inference process. However, inference engines must follow total strategies, not partial ones. The selection between different total strategies satisfying a partial strategy is a control task, not an inference task, and is therefore the province of the control procedure.

When we speak of a problem solver pursuing a given strategy this does not necessarily mean that all steps in the strategy will be performed. If one of the steps in the strategy succeeds in providing a solution to the problem there is no reason for the problem solver to continue with any of the remaining steps in the strategy. Thus, for our simple example, if we chose the strategy $\langle 1, 3, 5, 4, 2 \rangle$ the steps 4 and 2 would never be executed since step 5 solves the problem. A strategy should therefore be thought of as a preference ordering on possible inference steps rather than a mandate to exhaustively perform all steps in the sequence.

Any strategy that includes all possible inference steps for a problem is called an *exhaustive* strategy. In the example above, any strategy containing all five steps would be exhaustive. If a strategy has the same probability of success as an exhaustive strategy it is called a *complete* strategy. In other words, a complete strategy is an exhaustive strategy with some of the redundant or superfluous parts removed. If a solution to a problem exists within a search space, any complete strategy will solve the problem. All exhaustive strategies are therefore complete, but a complete strategy need not be exhaustive. In the example above, the strategies $\langle 1, 4 \rangle$, $\langle 3, 5 \rangle$ and any strategies containing either of these as subsequences are complete. We refer to the strategies $\langle 1, 4 \rangle$ and $\langle 3, 5 \rangle$ as *minimal* complete strategies.

2.2.3 Control

Controlling inference means guiding the inference process. More precisely, control is the limitation of the set of inference steps allowed at any stage of the inference process. With no control whatsoever, the inference steps allowed at each stage are limited only by the partial ordering between steps and their necessary predecessors. Any fact or goal that follows from existing facts and goals could be generated, and stands an equal chance of being generated. At the other extreme is *total* control, where only one possible inference step is permitted at each stage of the inference process. In this case inference is limited to a chosen subset of the set of all possible inference steps, and a total ordering is imposed on this set. Total control amounts to the imposition of a total strategy on an inference procedure.

2.3 The Utility Principle

Suppose we have a problem to solve and several different inference steps are possible at a given point in time. How do we decide which of the different steps to try? The criteria we will adopt is a very simple and intuitive one; choose the step that leads to a solution in the quickest and easiest fashion. This is the basic principle of utility.

Utility Principle: *The best step for an inference system to take is the step that will result in the lowest expected cost for solving the problem.*

Although it may not be obvious at first, the utility principle has an assumption of completeness built into it. If inference steps are selected on the basis of this principle, an inference engine will never give up on a problem so long as possibilities remain that have non-zero probability of success. An inference engine using the utility principle will therefore pursue a complete strategy.

Corollary 2.1 *The best strategy for an inference system to follow is the complete strategy having the lowest expected cost.*

We will refer to the complete strategy with the lowest expected cost as the *optimal* strategy for a problem. Note that this concept of optimality is not fixed with respect to a particular problem. Its definition is relative to the information and method used for estimating expected cost.

To apply the utility principle, the *promise* of each possible inference step must be evaluated (i.e. the likelihood that the step will lead to a solution in a quick and easy fashion). This is not a simple matter. As pointed out in Chapter 1, local criteria, such as the complexity of the expressions involved in a step, do not provide an accurate indication of either the expense, or the likelihood of finding solutions to a problem. For example, a goal expression having only a single conjunct might lead to complex and unsolvable subproblems. Likewise, a goal expression with many conjuncts might admit an immediate solution from the database.

A good evaluation of the promise of an inference step requires an evaluation of the expected cost and probability of success for all strategies that begin with that step. The cheapest complete strategy is actually found as a side effect of this process. In fact, it turns out to be easier to do the latter, than the former. As a result, our approach will be to search for the cheapest complete strategy, rather than to explicitly evaluate the promise of the different possible inference steps for a problem.

One possible objection to this approach is the requirement of completeness. It might be preferable, in some cases, to adopt strategies that are less than complete. For example, we might want to rule out steps having a low probability of leading to a solution, or rule out strategies where the expected cost exceeds some established upper bound.

This objection has some merit (see [Rai70]). However, to overcome it we would have to abandon or radically alter the basic utility principle. This would dramatically increase the complexity of the analysis involved in finding the best inference step. In addition, it is not clear precisely what factors to consider. As a result, only the basic utility principle will be considered in this dissertation.

2.4 Details of the Approach

2.4.1 Utility Calculations

In order to find the cheapest complete strategy we need to calculate the expected cost and probability of success for different strategies. Using simple utility theory it is possible to compute these quantities from basic information about the expected cost and local probability of success for the individual steps involved in a strategy. This requires some way of estimating the local probability of success for the different inference steps involved in a strategy. One way of doing this would be to actually perform the inference steps in the strategy. The problem is that it is far too expensive to generate and try all of the inference steps for a specific problem.

Another way of calculating the probability and expected cost of strategies is to make an *averaging assumption.* With an averaging assumption, instead of considering the specific strategies available for solving a particular problem, we consider the abstract or *generic* strategies for the class of problems having the same form. For example, instead of generating all of the specific strategies for the goal $Sibling(Rob, s)$, we consider the generic inference steps and strategies that apply to any problem of the form $Sibling(P001, s)$ where $P001$ is some arbitrary person. If the rule

$$Brother(s, b) \Longrightarrow Sibling(s, b)$$

is in the database, the generic inference step of reducing the *Sibling* goal to a *Brother* goal always applies. If there are brotherhood facts in the database, the inference step of looking up a *Brother* subgoal in the database also applies, but has only a limited chance of succeeding (since the constants in the subgoal may or may not match any of the specific facts in the database).

Using simple probability theory we can calculate the likelihood that such generic backward inference steps will succeed over the class of goals of a given form. Likewise, we can calculate the likelihood that generic forward inference steps will succeed over the class of facts of a given form. Using this information, basic utility theory allows the calculation of the probability of success and expected cost for generic strategies applied to problems of the given form.

The big advantage of averaging assumptions is that they allow the same expected cost calculations to serve for many different problems, or in some cases, many times in the same problem. In both cases the overhead of calculating expected cost is amortized over many different cases. We will make use of such averaging assumptions to assess the promise of different strategies in the case studies of Chapters 5 and 6.

2.4.2 Searching the Strategy Space

Using the approach of the previous section we could, in theory, compute the expected cost for all complete strategies and compare to find the cheapest one. Unfortunately, the space of complete strategies is immense for any non-trivial problem. In fact, it is much larger than the space of possible inference steps. For the kinship example, introduced in Chapter 1, there are more than 10^9 complete strategies containing just backward inference steps. A complete search of the strategy space to find the cheapest complete strategy is therefore rather intractable.

This situation can be improved somewhat. The cost of a strategy is always less than the cost of any longer strategy that contains it as a beginning substrategy. This means that best-first search can be used to search the space of possible strategies by building strategies of increasing length, and using the expected cost of candidate strategies as the evaluation function. Beginning with the null strategy, candidate strategies of greater and greater length are constructed by adding additional inference steps onto the end. The candidate strategy with the lowest cost is always expanded next. When the strategy with the lowest expected cost turns out to be a complete strategy, no further search is necessary, the optimal strategy has been found.

Although best-first search provides some improvement over complete search of the space of possible strategies, it is still not good enough. Suppose, for example, that all inference steps have the same cost. A best-first search of the strategy space would still involve at least as much search as a breadth-first search of the inference space. To make a search of the strategy space practical, we must find ways of significantly reducing the size of the space so that only a relatively small number of complete strategies need be considered and compared.

2.4.3 Reduction of the Search Space

One way of reducing the size of the strategy space for a problem is to eliminate portions of the inference space that are either superfluous or redundant. If a portion of an inference space is superfluous, it can be eliminated without sacrificing completeness, and without sacrificing optimality. From the standpoint of utility theory, superfluous portions of a search space are portions that have *zero* probability of contributing an answer, and hence *zero* utility.

For redundancy, the matter is more complicated. Initially, it might seem that any redundant portion of a search space could also be eliminated, yielding a corresponding reduction in the strategy space. Although one can eliminate redundant portions of a search space without sacrificing completeness, optimality may be sacrificed. This is because a redundant portion of the search space may provide the quickest solution to a problem. For example, in trying to find the phone number

of a colleague, a quick glance in the address book is warranted before resorting to long-distance information, even though the former is redundant with the latter.

Despite this, the recognition of redundancy does help to reduce the size of the strategy space.

Theorem 2.2 *Let s be a set of inference steps such that no step in s has any successors not in s. If the steps in s are redundant with another set of steps s', any strategy where steps in s follow all steps in s' is not optimal.*

Proof *Since s is redundant with s', once the steps in s' are performed, there is no value in performing any of the steps in s. Any such strategy is therefore suboptimal.* □

Theorems like the one above are referred to as *reduction theorems*, since they give conditions for eliminating portions of the space of possible strategies.

As Kowalski has pointed out [Kow70], redundancy is very common in inference spaces. For example, all of the different possible orderings for solving a conjunction are mutually redundant. Likewise, forward inference steps are usually redundant with their backward counterparts and vice-versa. But apart from these simple cases, establishing redundancy and superfluousness in a search space is hard. The case studies in Chapters 3 and 4 deal with specific problem solving situations where redundancy can be demonstrated. In a few special cases, redundancy can be recognized by purely syntactic means, as in the case study dealing with repeating goals (Section 4.3). However, in general, proving that portions of a search space are redundant or superfluous involves using information about the contents of the system's database as well as information about the properties of the relations involved. An example of such proofs can be seen in the case study dealing with divergent inference (Section 4.4).

Proofs of redundancy and superfluousness are the means for recognizing that non-exhaustive strategies are complete. If no such results can be obtained for a problem, the only complete strategies are those strategies that are exhaustive. Thus, recognizing redundant and superfluous portions of a search space is often a critical part of controlling inference (as the case studies will illustrate).

2.4.4 Reduction of the Strategy Space

Although we have already introduced one reduction theorem for redundancy, there are many other reduction theorems that apply only to more specific problem solving situations. Several of these are developed and discussed in the case study on ordering conjunctions (Chapter 5) and the case study of backward inference (Chapter 6). Using information obtained with the help of the averaging assumption, these

theorems can be used to reduce the strategy space to a tractable size. In a few special cases they even allow the optimal strategy to be constructed incrementally without search.

2.4.5 Sources of Information

Three different sources of information are useful for evaluating the potential efficacy of inference strategies:

1. Information about the system's database
2. Information about the world
3. Information about what kinds of problems the system is expected to solve and their relative frequencies

Information about the system's database includes information about what relations are represented, the number of facts in the database that will match any given form, information about the completeness of the database for a given relation or subject area, and information about how the database will change (what sorts of new facts can be expected and their relative frequencies). For a simple kinship problem, it is useful to know how many $Brother, Mother$, and $Father$ facts there are in the database. It is also useful to know that the database is incomplete for these three relations, and that these particular familial relationships are *invariant*. (Additional facts involving these relationships may become true through birth, but the existing ones will always remain true.) All of this is *meta-* information about the facts in the system's database. Such information is usually not explicitly represented in problem solving systems, but there is no reason why it cannot be. The information is readily available and given the language conventions of Section 2.1 it is possible to state it precisely.

The second kind of information, information about the world, includes things like the sizes of sets. For example, in dealing with the kinship example it is useful to know that a person has exactly two parents and has somewhere between zero and twenty siblings, usually three or fewer. Also useful is knowledge of the domains of each argument for a relation. In the kinship example all relationships have domains of people, males, or females. Relations may also have special properties like functional dependence among certain arguments, transitivity, symmetry, reflexivity, or temporal invariance. For example, both the mother and father relations are functions, the sibling relation is symmetric and the ancestor relation is transitive. For relations with a functional dependence among the arguments there may be many additional properties such as associativity, commutivity, monotonicity, boundedness,

continuity, etc. This information can all be expressed as second order statements.[2]

Finally, there is information about what sorts of problems the system will be expected to solve, and the frequency with which it will be expected to solve them. If this information is not available, then the system must either make randomness assumptions or must be prepared to make different control decisions depending on the characteristics of each given problem.

2.5 Summary and Final Remarks

The intent of this chapter has been to establish a basis for understanding how the case studies of Chapters 3 through 6 fit together. We have adopted a utilitarian approach, that the best inference step to perform is the one that will lead to the lowest expected cost for solving the problem. This has lead to a view of the control process as a search of the space of complete strategies to find the cheapest complete strategy. Basic utility theory, together with averaging assumptions, provide the means of computing the expected cost and probability of success for generic strategies from estimates of the local probability of success and expected cost of individual generic inference steps. What makes all of this tractable is reduction theorems. It is these theorems that allow us to eliminate large portions of the strategy space from consideration.

There are several important differences between this general approach and most previous work on controlling inference. First, there is an overall difference in perspective. Control is often regarded as a process of pruning and ordering of a search space. Instead, we have viewed control as a process of searching and pruning of the space of possible strategies.

Second, we have chosen to accept higher control cost in exchange for more accurate evaluations of the promise of possible inference steps. This results in lower inference cost, but higher control cost.

Third, we have taken a conservative approach towards pruning of the search space. In AI systems there has often been a concentration on pruning rather than ordering of the search space. Completeness is usually sacrificed in the process. On the other hand, if a good job of ordering is done, inference steps that are unlikely to lead to solutions will naturally be postponed until late in the strategy. This makes overzealous pruning largely unnecessary. Only superfluous and redundant portions of a search space can be pruned without sacrificing completeness. Even then, the pruning of redundant portions of a search space can result in sacrificing the optimal strategy. Pruning the search space is dangerous. Pruning the space of possible strategies is much less so.

[2]Meta-level statements can be used instead if there is a desire to avoid second order statements.

A final point of difference is that heuristics used for controlling search are usually empirically derived. The methods developed here have a sound statistical basis. This does not mean that many heuristics do not have such a basis, only that this basis has rarely been carefully explicated or examined.

Chapter 3

The Number of Answers[1]

3.1 Motivation

Consider a database containing the facts

$$Parent(Jerry, Edmund)$$
$$Mother(Jerry, Priscilla)$$

and the axioms

$$Mother(c,p) \Longrightarrow Parent(c,p)$$
$$Father(c,p) \Longrightarrow Parent(c,p)$$
$$Child(p,c) \Longleftrightarrow Parent(c,p)$$
$$Sibling(c,d) \wedge Parent(d,p) \Longrightarrow Parent(c,p)\ .$$

Suppose the inference procedure is then asked to solve the problem of finding the natural parents of an individual,

$$find\ all\ y\colon\ Parent(Jerry, y)\ .$$

One solution would be discovered by database lookup and the second would be discovered using the *Mother* axiom. But the problem solver wouldn't stop here, it would continue to use the other rules to try to find more of Jerry's parents. A person has only two natural parents, however, so search can and should be halted once two distinct answers have been found.

This problem occurs any time all of the answers are needed for a problem. Such problems are sometimes encountered directly as queries to intelligent systems. They are also encountered when conjunctions occur as subgoals of a query. Consider the simple conjunctive goal $A(x) \wedge B(x)$. Even though we might be seeking only a

[1]This chapter is a revised version of a paper that appeared as [Smi83]

single answer to the conjunction, we may have to locate many answers to $A(x)$ before finding one that will satisfy $B(x)$ (or vice versa). If it turns out that there aren't any answers to the conjunction, we will end up finding every answer to $A(x)$. It would be useful to know when all answers to $A(x)$ had been found, so that the search could be terminated.

For some applications it is possible to assume that all answers to a problem can be found by the same inference path. In this case, when one or more answers are discovered, the remainder of the space is redundant and can be discarded. But this assumption is not reasonable for all applications, as illustrated by the Jerry Brown example. In this case, even after one answer has been found in the database, additional search is required to find a second answer. If the search space is large it may not be practical for the inference procedure to exhaustively search the space to make sure it has found all of the answers.

The solution to this problem is for the problem solving system to use information about the number of solutions to problems so that it can recognize when all of the answers have been found. Once all of the answers for a problem have been found, the remainder of the search space is redundant and can therefore be discarded (by Theorem 2.2).

In the next section we show how to specify, automatically derive, and automatically maintain information about the number of solutions to a problem. Section 3.3 is about the rather surprising difficulty that arises when other pruning strategies are used in conjunction with the techniques of Section 3.2. Related issues and the results of preliminary testing are discussed in the final section.

3.2 Using Knowledge About the Number of Answers

Let the meta-level functional expression $NumberOf(v, p)$ refer to the number of solutions for the desired variables v to the goal proposition p. Using this function it is easy to state that the problem of finding Jerry's parents has only two solutions.

$$NumberOf(\text{“}y\text{”}, \text{“}Parent(Jerry, y)\text{”}) = 2$$

More generally, we can state that all *Parent* problems of this form have only two solutions.

$$NumberOf(\text{“}y\text{”}, \text{“}Parent(\chi, y)\text{”}) = 2$$

Using this information a problem solver could determine how many solutions such a problem has, and could therefore halt reasoning when all answers had been found.

It would be a nuisance if we had to specify such meta-information for every problem a system might encounter. Fortunately, such information can be derived

from simple base-level knowledge about the sizes of sets. If σ is the set corresponding to the solutions to a proposition ϕ, the number of solutions to the proposition will be the cardinality of the set σ.

$$(\beta \in \sigma \iff \phi|_\beta) \Longrightarrow Numsol(\phi) = \|\sigma\| \tag{3.1}$$

As an example, using the axiom given above, and cardinality information like

$$\begin{array}{ll} \|Parents(x)\| = 2 & y \in Parents(x) \iff Parent(x,y) \\ \|HeartChambers\| = 4 & x \in HeartChambers \iff Chamber(x, Heart) \\ \|ShipsInBoston\| = 14 & x \in ShipsInBoston \iff Ship(x) \\ & \quad \wedge \exists d(Dock(x,d) \wedge Loc(d, Boston) \end{array}$$

a problem solver can infer the number of solutions to each of the corresponding problems, enabling it to halt inference when all of the solutions have been found.

There are also two special cases where information about the number of solutions to a problem can be inferred without domain specific information about set sizes. If the problem is to find out whether or not a proposition is true and the requestor doesn't care about any of the variable bindings, there is (at most) one solution to the problem.

$$NumberOf(\emptyset, p) \leq 1 \tag{3.2}$$

A common case of this is when the proposition p is a ground clause (contains no variables). Functional expressions also have at most one solution. This fact can be expressed as

$$Function(r) \wedge Variable(v) \wedge Ground(\vec{x}) \Longrightarrow \| \{v : r(\vec{x}, v)\} \| = 1\,. \tag{3.3}$$

3.2.1 Gathering Cardinality Information Dynamically

There are many instances where the addition of a priori knowledge about the sizes of sets is sufficient to solve the problems in the domain of interest. For example, in a system modelling human physiology, the number of heart chambers, the number of bones in the hand, and the number of major arteries are all unlikely to change. But there are also circumstances where it is either inconvenient or impossible to supply that knowledge. It would be unreasonable to require that a system builder supply the cardinality of several thousand sets, each having thousands of members. Consider, for example, the number of employees for each of the departments in a large corporation. It would be more than inconvenient if this counting had to be done monthly, weekly, or daily as the system's data base was upgraded.

The situation is worse when a problem solver is dealing with a constantly changing world. Consider an expert system that serves as an intelligent interface for an

operating system. The *intelligent agent* must accept requests from the user, make appropriate plans for their realization, perform the necessary system specific actions, and report results to the user. Knowledge about the number of tape drives and printers attached to the machine is (relatively) static and can be specified a priori. But the number of files in a given directory changes frequently and cannot be specified beforehand.

In this volatile environment there is still much that can be done. For problems where the cost of solution is high and the problem is often repeated, it is worthwhile to *cache* information about set sizes. To illustrate the technique, suppose that an intelligent agent has the task of finding all of the Scribe files on some particular directory (perhaps as a subproblem of transporting them to a new machine). The query is

$$\textit{find all } f\textit{: } Format(f, Scribe)\ .$$

For purposes of illustration, assume the intelligent agent believes that a file contains Scribe text if the file's extension is MSS, or if the file contains an @MAKE statement at the beginning. This is stated formally as

$$\begin{aligned} &Name(f,n) \wedge Extension(n, MSS) \Longrightarrow Format(f, Scribe) \\ &Contents(f,c) \wedge Word(1,c,@MAKE) \Longrightarrow Format(f, Scribe)\ . \end{aligned}$$

In this case, it is not reasonable for the system to have a priori knowledge about the number of Scribe files in a particular directory.

It would, however, be simple enough for the system to *cache* this information after the problem has been solved once. Suppose, then, that such a query is posed to the intelligent agent and that it exhaustively hunts through all of the files in the directory. Five Scribe files are found, three with the file extension MSS and two that do not have the usual extension, but contain @MAKE statements at the beginning. After counting all of the answers, the problem solver can cache the knowledge that there are exactly five such Scribe files (and hence five answers to the problem).

$$\begin{aligned} &\|ScribeFiles\| = 5 \\ &x \in ScribeFiles \Longleftrightarrow Format(f, Scribe) \end{aligned} \tag{3.4}$$

If the query is posed again, (perhaps as a subgoal of some other request) cached information about the number of Scribe files (3.4) will stop the problem solving process once those five Scribe files are found. If the answers to the problem are also cached, the solutions to any subsequent query would be found immediately in the data base. In this case, the problem solver would not have to dissect a single filename or look at the contents of a single file. Counting the number of solutions to a problem, inferring the size of the set, and caching this knowledge permits a quicker response to future queries, just as the presence of a priori knowledge about set size would.

As suggested in the previous section, instead of caching the cardinality of the set of Scribe files, it is also possible to cache the fact that each of the five files is a member of the set of Scribe files, and that there are no other such files. The difficulties of keeping this knowledge accurate (the subject of the next section) are the same for either of these representations.

3.2.2 Maintaining the Accuracy of Cardinality Information

There is only one problem with the scheme of Section 3.2.1, succeeding requests to the intelligent agent might cause files to be added, deleted or modified. Cached knowledge about set sizes could therefore become invalid, just as cached solutions might become invalid. A standard solution to this dilemma is to keep justifications for cached information, and use *truth maintenance* [Doy79] to remove assertions that become invalid as a result of other actions. The same technique can be used to keep track of cached knowledge about set sizes, provided that justifications are kept for these statements. In the example of the previous section, the statement "there are five Scribe files" rested on the assertions that there are three files with the extension MSS, and two files that begin with @MAKE statements.[2] These statements, and their justifications can be recorded formally as indicated below.

$$\|ScribeFiles\| = 5$$
$$\|MssFiles\| = 3$$
$$\|FileNamePairs\| = 237$$
$$\|@MakeFiles\| = 2$$
$$\|FileContentsPairs\| = 237$$

$$Justification(\text{“}\|ScribeFiles\| = 5\text{”}, \{\text{“}\|MssFiles\| = 3\text{”}, \text{“}\|@MakeFiles\| = 2\text{”}\})$$
$$Justification(\text{“}\|MssFiles\| = 3\text{”}, \{\text{“}\|FileNamePairs\| = 237\text{”}\})$$
$$Justification(\text{“}\|@MakeFiles\| = 2\text{”}, \{\text{“}\|FileContentsPairs\| = 237\text{”}\})$$

[2]There are other dependencies as well, but these are the two important ones for most cases. The view presented here is a simplified one, though it is entirely possible to implement a system that performs in this manner. In reality, the cardinality statement for a set is derived from the meta-level information about the number of solutions using (3.1). This meta-level proposition, in turn, has many dependencies. Among them is the statement that there are no more possible inference paths that might contribute anything to the solution of this problem. In general, any meta-level statement about the kinds of things in a database may be invalidated by updates to the database. This is true because such statements depend implicitly on facts of the form $InDatabase(p)$ or $\neg InDatabase(p)$. In general, when p is added to or removed from a system's database, the meta-level must be informed that $InDatabase(p)$ or $\neg InDatabase(p)$ is now true, and be permitted to make the appropriate inferences.

Given these justifications, knowledge about set sizes can be kept accurate when the database is changed. For example, suppose that the intelligent agent is instructed to delete the file named BOOK.MSS. Assuming that the intelligent agent was caching solutions as well as information about set size, its database might contain the following propositions:

$Name(File37, BOOK.MSS)$
$Extension(BOOK.MSS, MSS)$
$Format(File37, Scribe)$

$Justification(\text{"}Format(File37, Scribe)\text{"}$
$\{\text{"}Name(File37, BOOK.MSS)\text{"}, \text{"}Extension(BOOK.MSS, MSS)\text{"}\})$

After performing the deletion the intelligent agent must update its database to reflect the effects of the action. The fact $Name(File37, BOOK.MSS)$ is removed, and then, using truth maintenance, the fact $\text{"}Format(File37, Scribe)\text{"}$ is removed since it depends upon the former. In addition, since the fact $Name(File37, BOOK.MSS)$ has been removed, the fact about the number of filename pairs is no longer valid, and must be removed. By removing this cached cardinality assertion, the entire tree of cached cardinality information unravels, and the statement that there are five Scribe files will be removed.

In general, all that is required (in addition to saving justifications and performing truth maintenance) is the inference that a cached statement about the size of a set must be removed any time a proposition matching (unifying with) the set definition is added to or removed from the database.

This methodology is not without cost. While the additional reasoning required is minimal, caching all of the set size information and the associated justifications does require storage. It is, however, often much less than the storage overhead of caching base-level facts and their justifications. If a problem has two hundred answers, caching the results demands storing at least two hundred propositions and two hundred justifications (many more if intermediate results are derived and cached). In contrast, only one cardinality assertion and its justification need be cached for each subproblem encountered.

3.2.3 Invariance

As a practical matter, it is often possible to cut down the overhead of maintaining information about set sizes by specifying *invariance* for static quantities. In the case of the intelligent agent the number of tape drives usually doesn't change. Thus, we could specify

$$Invariant(\text{"}\|TapeDrives\| = \nu\text{"}) .$$

If a cached statement is *invariant* (will not change over time), there is no need to keep its justification around. If a problem solver dynamically calculates the size of a set, but finds that there is an invariance statement (like those above), the system can cache the statement and use it without storing elaborate justifications.

It is also possible to make more general statements about invariance. For example, the elaborate justification mechanism could be completely defeated by asserting

$$Invariant(p) .$$

Probably more useful is to specify interesting classes of expressions that are invariant. For example, we can state that the number of hardware peripherals is invariant.

$$Subsumes(Peripheral, \rho) \Longrightarrow Invariant(\text{“}\| \{x : \rho(x)\} \| = \nu\text{”})$$

Given this, it is only necessary to specify that something is a peripheral (e.g. $Subsumes(Peripheral, Printer)$) in order to get the property of invariance.

The use of invariance is not limited to the techniques of this paper. Stating invariance can be advantageous whenever a system is dealing with a changing world. Under these circumstances much of the overhead of *truth maintenance* (or whatever techniques are used to deal with a changing world) can be eliminated. It has been mentioned here because it is often applicable to statements about set size, and because the justifications and maintenance for these statements are quite messy. Judicious use of these shortcuts can help to keep down both the computational and storage overhead involved in this scheme. The more elaborate techniques of Section 3.2.2 will then only come into play for those cases where they are truly needed.

3.3 Interaction with Other Pruning Strategies

> *When we try to pick out anything by itself we find it hitched to everything else in the universe.*
>
> *My First Summer in the Sierra – John Muir*

There are some subtle and surprising interdependencies that arise between statements about the number of solutions, and other control methods used to eliminate redundant portions of a search space. We will first give an abstract characterization of the difficulty and then illustrate with a specific example.

Suppose that a problem solver is attempting to find the solutions for a goal expression P. In doing so, it generates the subgoal Q, which generates the subgoal R, as shown in Figure 3.1. Further suppose that the subgoal R is redundant

with P and therefore cannot possibly contribute any new solutions to P. Consider what happens if the subgoal R is pruned, and the system is keeping track of, and caching, the number of solutions to each of these subgoals (as in Sections 3.2.1 and 3.2.2). Suppose that there are three solutions to the problem R, five solutions to the problem Q, and ten solutions to the original problem P. The number of solutions calculated (and cached) for the problem P will still be correct, because the unique solutions have been eliminated by pruning. But consider what number will be calculated for the subproblem Q. If all three of the solutions to R contribute to unique solutions for Q, only two of the solutions to the problem Q will be found, rather than all five. This number is wrong, and if cached, will cause difficulties when the problem Q is encountered again, either in vacuo, or in the context of some other problem.

As a specific example consider two simple facts about familial relationships,

$$Child(x,y) \wedge Male(x) \Longleftrightarrow Son(x,y)$$
$$Child(x,y) \wedge Female(x) \Longleftrightarrow Daughter(x,y),$$

together with the simple database,

$$Son(Stacey, Martha)$$
$$Son(Brooks, Martha)$$
$$Daughter(Jan, Martha)\ .$$

Suppose that the query

$$find\ all\ x\colon\ Son(x, Martha)$$

is posed to the system. After discovering the two answers immediately in the database ($Brooks$ and $Stacey$), the subgoal

$$Child(x, Martha) \wedge Male(x)$$

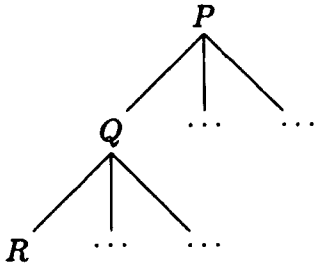


Figure 3.1: Simple Subproblem Tree

is generated. Backward reasoning on the *Daughter* axiom produces the answer *Jan* for this subgoal. But the *Son* axiom is also applicable to this subgoal, and would produce two additional answers to the subgoal. But because of the fact that the primary objective is to find Martha's sons, and the first axiom has just been applied in one direction, there is absolutely no point in turning around and applying it in the opposite direction (see Chapter 4). It won't lead to any new solutions. As a result, that line of inference can be discarded, and only one solution is produced for the subproblem of finding Martha's children.

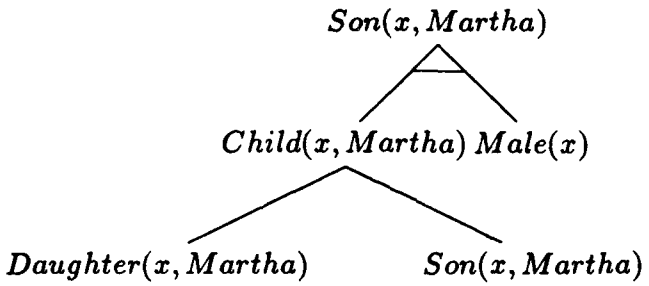


Figure 3.2: Subproblem Tree for Kinship Problem

Ordinarily, this would cause no harm. All of the answers to the original query are found, and if any intermediate results are cached they will still be correct, but perhaps incomplete. Consider, however, what happens if we blindly apply the methods of Section 3.2.2. We would end up caching the following two facts:

$$\|MarthasSons\| = 2$$
$$\|MarthasChildren\| = 1$$

The first is correct, but the second is wrong. Martha has three children, not one. This happens because a portion of the search space was pruned due to consideration of something outside the context of the subgoal $Child(x, Martha)$.

A solution to this difficulty is to introduce an additional argument in the $NumberOf$ relation. If any contextual information is used to help solve this subproblem, it must be recorded in the additional argument. This contextual information is, in effect, represented by the *justification* of the meta-level conclusion that a portion of the search space can be eliminated. The details of this justification are dependent upon the specific structure of the problem solver involved, as well as the particular technique of meta-level pruning.

As a practical matter, it is possible to simply use a T for the context argument in those cases where additional pruning has taken place. This would mean that cached statements (ones with T's) are not useful for estimating the number of solutions

when the subproblem is encountered again. Thus, their only purpose is to record the dependency tree for use in truth maintenance. This is probably not a large loss, since, even if full context arguments were kept, it is unlikely that a context argument would match up with any other except that of the original problem, whose results would be cached anyway.

Finally, two things should be noted. First, the cached information for the overall problem will be correct in any case. Second, the extra argument to $NumberOf$ statements will always be null if no other pruning takes place during the course of solving a subproblem. In this case, the proposition about the number of solutions would be the same as if the subproblem had been solved in vacuo.

3.4 Final Remarks

3.4.1 The Closed World Assumption

In the preceding sections, we assumed that a problem solver is capable of producing every solution to a problem. As a result, the theoretical number of solutions to a problem is the same as the number that the system can produce. This is known as the closed-world assumption.

The closed-world assumption is not essential to the techniques presented here. The theoretical number of solutions is an upper bound on the actual number that a problem solver can produce and remains useful in any case where the system can produce all of the answers to a problem. Cached information, however, is information about the actual number of answers a problem solver can produce. To distinguish these two concepts, a new relation, $ActualNumberOf$, can be introduced, to refer to the number of solutions to a problem that can be found by a problem solver.

$$ActualNumberOf(v,p) \leq NumberOf(v,p)$$

The actual number would then be used for terminating inference instead of the theoretical number.

In systems where the closed world assumption is not valid there is additional advantage to having both kinds of information available. Information about the actual number of solutions permits inference to be stopped, while theoretical information allows a system to warn the user when it cannot produce all of the answers to a problem.

3.4.2 Interaction with Ordering Strategies

In Section 3.3 we considered the negative effect that other pruning strategies have upon keeping accurate cardinality information. In contrast there is a positive effect that occurs when search space ordering is combined with the methods presented here. Ordinarily, when a system is asked to find all of the solutions to a problem, ordering of the search space is of no benefit. The entire space must be searched anyway. However, when the number of solutions is known, ordering of the search space becomes valuable. If all of the answers can be collected rapidly (by searching promising branches of the space first), more of the space will be pruned by a recognition that all of the solutions have been found. Conversely, the utility of knowing the number of solutions depends on discovering all of the solutions before the space is exhausted. For problems in which all the solutions are desired ordering is of no use without knowing the number of solutions, just as knowing the number of solutions is of no use under the worst possible ordering.

3.4.3 Performance

The methods described in this chapter have been implemented in an experimental version of the MRS system [Rus85]. One application where these methods have been used is in a system for causal simulation of renal physiology [Kun84]. Without any information about set sizes, difficult queries of the model usually required on the order of fifteen minutes of CPU time (on a DEC 20/60). When knowledge about the sizes of important sets and several invariance statements were added, the total CPU time for difficult queries was reduced by fifty to ninety percent, depending on the particular query. While invariant set sizes were cached, the more elaborate storing of justifications outlined in Section 3.2.2 was not complete at the time. The effort was also limited by the inability to express certain complex invariance statements in that version of MRS. These deficiencies have since been remedied, allowing additional performance improvement using only these elementary techniques.

A second application of the techniques presented here arose in the development of a simple income tax consultant [Gen83]. During a consultation, the system needs to make inferences about familial relationships. Because of the many different familial relationships involved (e.g. Mother, Father, Aunt, Uncle, Sister, Brother, Daughter, Son, Sibling, Child, Parent), the search space for even the simplest queries was extremely large. For example, a simple query to determine all of the siblings of a given person generated hundreds of subgoals (fifteen pages of hardcopy trace) and took nearly fifteen minutes of real time to produce the answers. Circular reasoning was pruned as in the example of Section 3.3. The addition of information about the number of parents a person has, and knowledge that *Mother* and *Father* are function symbols, reduced the number of subgoals by a factor of three, and reduced

the run time by a similar factor. Use of the full techniques described in Sections 3.2.2 and 3.3 resulted in an additional factor of two reduction when running a problem for the first time. The reason for this speedup is that the same subgoals were often generated several times by different branches of the search tree. The first occurrence would cause the answers and knowledge about set sizes to be cached. Subsequent occurrences of the subproblem could then make use of the cached knowledge about set size.

Chapter 4

Recursive Inference

4.1 Introduction

4.1.1 Motivation

Consider a system for reasoning about circuits based on descriptions of circuit topology and the functional characteristics of circuit elements. Such a system might need to know that connection between terminals in a circuit is transitive,

$$Conn(x,y) \wedge Conn(y,z) \Longrightarrow Conn(x,z)\ , \tag{4.1}$$

where the proposition $Conn(x,y)$ means that the point x is electrically connected to the point y in the circuit. The problem with such facts is that they often result in infinite searches. Suppose, for instance, that we want to find *all* of the connections to some point A in a circuit. A portion of the backward AND/OR search tree for this problem is shown in Figure 4.1. Applying the transitivity rule to the query $Conn(A,z)$ results in the subgoal $Conn(A,y) \wedge Conn(y,z)$. The transitivity rule can be applied again to both of these conjuncts yielding the subgoals $Conn(A,y') \wedge Conn(y',y)$ and $Conn(y,w) \wedge Conn(w,z)$ respectively. Transitivity applies again to each of these four conjuncts, and so on. For this problem a backward inference procedure would apply the transitivity rule again, and again, and again until it runs out of storage, the user runs out of patience or money, or the machine crashes. We could eliminate the recursion in the connectivity problem simply by discarding all repeating subgoals, but we might sacrifice important answers in the process. For example, suppose that the database contains the facts,

$$Conn(A,B)$$
$$Conn(B,C)$$
$$Conn(C,D)$$

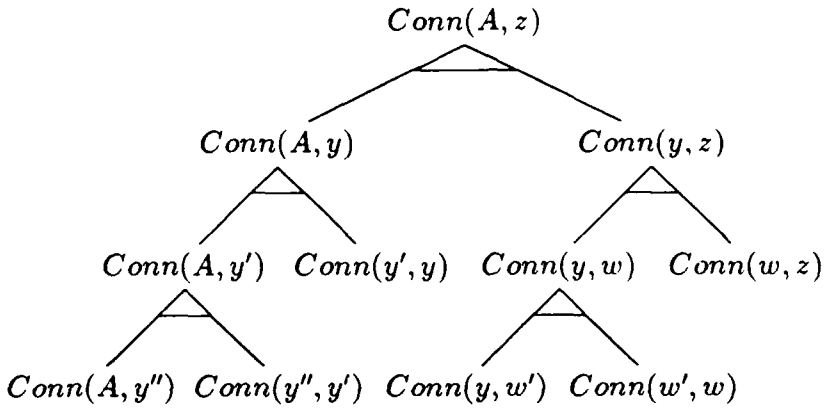


Figure 4.1: A portion of the backward search space for the goal $Conn(A, z)$

and the query is to find all answers to $Conn(A, z)$ as before. A database lookup on the goal expression yields the solution $z = B$. The remaining two solutions, $z = C$ and $z = D$ both require an application of the transitivity axiom. But after one application of the transitivity axiom, we have a subgoal containing the conjunct $Conn(A, y)$, which matches the original goal. This repeating subgoal is the only possible subgoal for our problem. If it is eliminated, neither of the remaining two answers will be found.

For this example, if we were to allow recursion only one level deep, all of the answers could be found. A solution $y = B$ to the subgoal $Conn(A, y)$ is found using only database lookup. Substituting this binding into the remaining conjunct gives the subgoal $Conn(B, z)$. The solution $z = C$ is found to the original goal, $Conn(A, z)$, by database lookup on this subgoal. Next the transitivity axiom is applied again to this subgoal yielding the subgoal $Conn(B, w) \wedge Conn(w, z)$. Again permiting a single level of recursion, the solution $w = C$ is found by database lookup on the first clause leaving us with the subgoal $Conn(C, z)$. A database lookup on this subgoal yields the final answer $z = D$.

Unfortunately not all recursion can be eliminated this simply. In general, an arbitrary number of levels of recursion may be required to find all answers to a problem. For example, consider an axiom of the form

$$P(x) \wedge R(x, y) \Longrightarrow P(y)$$

where we want to find all $P(z)$. A portion of the search space for this example is shown in Figure 4.2. Suppose that the database contains the facts

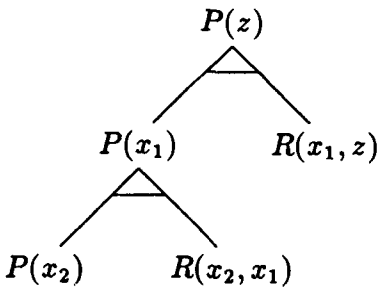


Figure 4.2: A portion of the backward search space for the goal $P(z)$

$$P(1)$$
$$R(1,2)$$
$$R(2,3)$$
$$R(3,4)$$
$$\vdots$$

The answer $z = 1$ will be found instantly by database lookup. The answer $z = 2$ requires one level of recursion. The answer $z = 3$ requires two levels, and so on. To find the n^{th} answer requires searching $n - 1$ levels deep in the recursive space.

In this chapter we develop methods for deciding how deep a problem solver must go in a recursive space. The amount of the space that must be examined depends, in general, upon the form of the recursion, as well as on the facts that are present in the system's database. But before proceeding, we need to consider some other alternatives for dealing with recursion.

4.1.2 False Hopes

It might seem that there are several easy solutions to this problem, such as using breadth-first search, using selective forward inference, or reformulating the troublesome axioms to make the problem go away. In fact, as we will demonstrate in this section, none of these measures work very well.

Breadth-first Search

Suppose we were to use breadth-first search on a space like that in Figure 4.1. Using breadth-first search we are guaranteed to find any answer in the search space (eventually). Unfortunately this does not help if we are looking for *all* answers to a problem. If there is no information about when to stop looking for answers, the entire infinite space must be explored, and breadth-first search will never halt.

Even if we are only interested in a specific number of answers, breadth-first search will only halt if the space contains at least that many answers. For example, if we ask for only a single answer to a query, but it turns out there are no answers, breadth-first search will not halt. Thus, breadth-first search alone does not solve the problem of recursive inference.

Selective Forward Inference

In the connectivity example suppose that forward inference is selectively performed on all facts of the form $Conn(a, b)$ using the transitivity rule, and the transitivity rule is *not* used for backward inference. The recursion problem would then be eliminated, since the transitivity rule does not cause problems for forward inference. Unfortunately, there are several serious difficulties with this approach. First of all, even the selective use of forward inference can result in the computation and storage of many irrelevant facts. For the example above, forward inference would result in computation of the transitive closure of all connections in the circuit, even though we may only need to know the connections to a few. This would be unacceptable for circuits with high fan-out or fan-in, or for connections to common busses, power supplies, or grounds.

To make matters worse, it is not always possible to limit the use of forward inference to just those rules responsible for recursion.[1] Suppose that, in addition to the transitivity axiom, there is another axiom $P(x, y) \Longrightarrow Conn(x, y)$ in the database, along with the facts $Conn(A, B)$ and $P(B, C)$. If forward inference is confined to just the transitivity axiom, incompleteness can result. Since there is only one connectivity fact, no conclusions are drawn using forward inference on the transitivity axiom. If one were then to ask for all connections to A, the answer B would be found, but not the answer C. To fix this incompleteness, the axiom $P(x, y) \Longrightarrow Conn(x, y)$ must also be subject to forward inference. More generally, if an axiom is restricted to forward inference, all axioms that can be used in the proof of any of its premise clauses must also be subject to forward inference. Subjecting additional axioms to forward inference further increases the number of potentially irrelevant facts that must be computed and stored.

Another problem with forward inference is that it can also lead to infinite search spaces. Consider the rule for computing Fibonacci numbers:

$$\begin{aligned} x = Fibonacci(i-2) \wedge\ & y = Fibonacci(i-1) \wedge\ z = x + y \\ & \Longrightarrow Fibonacci(i) = z \end{aligned} \tag{4.2}$$

When two Fibonacci numbers are given to a forward inference procedure it would proceed to compute Fibonacci numbers forever. This rule can cause an infinite loop in either a backward or forward inference engine.

[1] Minker and Nicolas [MN83] and Reiter [Rei78b] have also pointed this out.

Thus, even the selective use of forward inference is not a good solution to the problem of recursive inference.

Reformulation

The technique of reformulation is one quite familiar to PROLOG programmers. It involves rewriting the facts available to the inference procedure so that the search space for the goal is no longer infinite, or so that the inference procedure will not discover the recursive portion. As an example of what we mean by reformulation, consider the troublesome transitivity rule (4.1) for circuit connections, together with the database

$$Conn(A, B)$$
$$Conn(B, C)$$
$$Conn(C, D)$$

Suppose we introduce a new relation, $IConn$, meaning "immediately connected". The database and transitivity rule can then be replaced by the facts

$$IConn(A, B)$$
$$IConn(B, C)$$
$$IConn(C, D)$$

and the two rules

$$IConn(x, y) \Longrightarrow Conn(x, y)$$
$$IConn(x, y) \wedge Conn(y, z) \Longrightarrow Conn(x, z)$$

For this example the answer $z = B$ can be found using the first rule. The reformulated transitivity rule can also be applied, yielding the conjunctive subgoal $IConn(A, y) \wedge Conn(y, z)$. By database lookup, the solution $y = B$ can be found to the first clause. Substituting this binding into the remaining conjunct leaves the subgoal $Conn(B, z)$. The solution $z = C$ can be found using the first rule. The reformed transitivity rule can also be applied again yielding the subgoal $IConn(B, y') \wedge Conn(y', z)$, and so on. The search space for this problem is shown in Figure 4.3. With this reformulation, we have eliminated the recursion on all left-hand branches of the tree. As a result, this reformulation does not lead to an infinite search, so long as the $IConn$ conjunct is solved before the $Conn$ conjunct.

There are several serious problems with reformulating information in this way. First of all, reformulations generally only work well for a few of the possible forms that a query might take. For example, the above reformulation works well for the query $Conn(A, z)$ but it does not work well for the query $Conn(x, D)$. On applying the reformulated version, we would get the subgoal $IConn(x, y) \wedge Conn(y, D)$. If the $IConn$ conjunct is expanded first we end up searching through all of the immediate connections in the circuit; a horribly inefficient process in a large circuit.

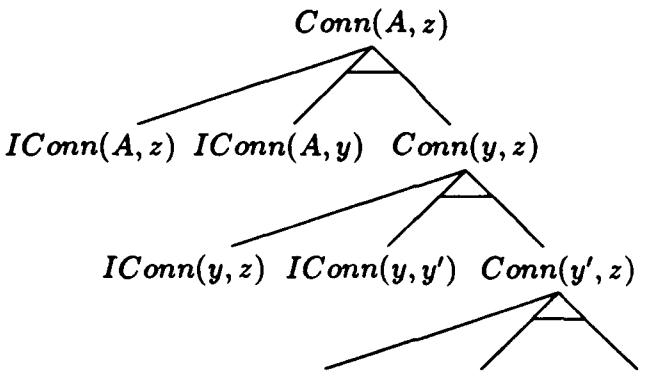


Figure 4.3: Reformulated search space for the goal $Conn(A, z)$

Alternatively, if the *Conn* conjunct is expanded first, we again end up with an infinitely repeating search space. The dual reformulation,

$$IConn(x, y) \Longrightarrow Conn(x, y)$$
$$IConn(y, z) \wedge Conn(x, y) \Longrightarrow Conn(x, z)$$

works fine for the query $Conn(x, D)$, but performs miserably for the query $Conn(A, z)$. Neither of these reformulations are reasonable if both kinds of queries are expected, as might be the case for an asymmetric relation. In general reformulations only work effectively for some subset of the possible queries covered by the original domain knowledge.

A second problem with reformulations is that they are often fragile. For the example above, adding the fact $IConn(B, A)$ to the database would again lead to a loop. As before, the subgoal $Conn(B, z)$ can be generated from the goal $Conn(A, z)$, but using the new fact, the subgoal $Conn(A, z)$ can be generated from the subgoal $Conn(B, z)$.[2]

A third problem with reformulation is that it can be an arbitrarily difficult programming task. Suppose that, in addition to the transitivity rule for connections, we also have the symmetry rule

$$Conn(x, y) \Longrightarrow Conn(y, x)$$

The reformulation now requires four facts, and another intermediate relation, *TConn*.

$$IConn(x, y) \Longrightarrow TConn(x, y)$$
$$IConn(x, y) \wedge TConn(y, z) \Longrightarrow TConn(x, z)$$
$$TConn(x, y) \Longrightarrow Conn(x, y)$$
$$TConn(x, y) \Longrightarrow Conn(y, x)$$

[2]We are indebted to an anonymous reviewer for this observation.

It is not so obvious that this reformulation covers all of the possible cases.

As an even more problematic example, consider the recursive rule that states that a person will be an albino if both his parents are albinos.

$$Albino(x) \wedge Parents(z) = \{x, y\} \wedge Albino(y) \Longrightarrow Albino(z) \tag{4.3}$$

Suppose that the query is to find all albinos.

$$find\ all\ z:\ Albino(z)$$

Expanding the parents conjunct first would result in an unacceptable search through all parent/child pairs. Alternatively, expanding either of the albino conjuncts first would result in an infinite repeating search space. By indulging in ***knowledge programming***, we could reformulate this rule so that depth-first backward inference results in an efficient search of the space for this query. As a first step we introduce the new predicate $GivenAlbino(x)$ to refer to those individuals given as albinos initially. The first two rules in (4.4) below state that any given albino is an albino, and that any n^{th} generation descendent of a given albino (along albino lines) is also an albino. We still need to define what it means for an individual to be of albino descent from a given albino. The third rule states that an individual is of albino descent from a given albino if the given albino is a parent of the individual, and the other individual's parent is an albino. The fourth rule simply expresses the transitive closure of this relationship, that an individual is of albino descent from a given albino if that individual is of albino descent from the givens albino children. The final two rules are for checking whether a given individual is an albino and are identical to the original albino rule (4.3).

$$\begin{aligned}
&GivenAlbino(z) \Longrightarrow Albino(z)\\
&GivenAlbino(x) \wedge AlbinoDescent(z, x) \Longrightarrow Albino(z)\\
&\\
&Parents(z) = \{x, y\} \wedge CheckAlbino(y) \Longrightarrow AlbinoDescent(z, x)\\
&Parents(w) = \{x, y\} \wedge CheckAlbino(y)\\
&\qquad \wedge AlbinoDescent(z, w) \Longrightarrow AlbinoDescent(z, x)\\
&\\
&GivenAlbino(z) \Longrightarrow CheckAlbino(z)\\
&Parents(z) = \{x, y\} \wedge CheckAlbino(x)\\
&\qquad \wedge CheckAlbino(y) \Longrightarrow CheckAlbino(z)
\end{aligned} \tag{4.4}$$

Performing depth-first backward inference on this reformulation results in forward inference from given albinos to their progeny, and backward inference at each step to verify that the other parent of the progeny is also an albino. Note that this backward portion of the inference is accomplished using the original albino

rule (4.3) (which is rewritten using a different predicate to distinguish it from our reformulated version). As with the connectivity example, this reformulation only works efficiently for the query $Albino(z)$, where one or more albinos are desired. It does not work well for checking whether a given individual is an albino.

From these examples, we can see that there are several serious disadvantages to reformulation as a method of controlling recursive inference. First, the resulting knowledge programs only work effectively for some subset of the possible queries covered by the original domain knowledge. Second, the programs may be fragile, in that additional data can reintroduce recursion. Third, it may be an arbitrarily difficult programming task to do such a reformulation. Finally, it is more difficult to understand, explain, and modify reformulations. Reformulation results in an implicit embedding of control information into the domain information. Instead of having facts about the domain and facts about control, the two are merged into knowledge-rich programs for a given interpreter. This has little advantage over building expert systems in more traditional programming languages like LISP or PASCAL. Many authors have argued against reformulation for exactly these reasons [McC68,Hay73,Dav80,Doy80,Cla83,GS85].

4.1.3 Definitions

So far we have relied on the readers intuitions and the examples to indicate what we might mean by the term recursive inference. We now give a precise definition.

Let the term *goal set* refer to the set of all conjuncts for a conjunctive goal in a search space. We say that one goal set g' is a *descendant* of another goal set g, if there is some sequence of goal sets beginning with g and ending with g', such that each goal set in the sequence is a subgoal of its predecessor. An *inference path* in a search space is a sequence of goal sets in the space such that each goal set in the sequence is an (immed..'e) subgoal of the preceding goal set. For example, the sequence

$$\begin{array}{l} \{Conn(A,z)\} \\ \{Conn(A,y), Conn(y,z)\} \\ \{Conn(A,y'), Conn(y',y), Conn(y,z)\} \\ \quad\vdots \end{array} \tag{4.5}$$

is an inference path for the connectivity problem.

We use the notation $p|_b$ to refer to the expression formed by substituting the variable bindings b into the expression p. Using this notation, an expression c' is said to be an *instance* of an expression c if there is a substitution (a set of variable bindings) b for the variables in c such that $c' = c|_b$.

Definition 4.1 *An inference path is* recursive *if there is an infinite subsequence* $\langle g_1, \ldots, g_i, \ldots \rangle$ *of the goal sets in the path and a distinguished clause* c_i *in each goal set* g_i *such that,*

1. c_i *is an instance of* c_{i-1}, *and*
2. c_i *is in the subset of* g_i *that are descendants of* c_{i-1}.

An inference procedure generating any recursive inference path is said to be involved in recursive inference.

As an example, consider the infinite inference path (4.5) for the connectivity problem. The conjunct $Conn(A, y)$ in the second goal set is an instance of the conjunct $Conn(A, z)$ in the first goal set. The descendants of $Conn(A, z)$ constitute the entire set $\{Conn(A, y), Conn(y, z)\}$, which contains $Conn(A, y)$. Likewise, the conjunct $Conn(A, y')$ in the third goal set is an instance of the conjunct $Conn(A, y)$ in the second goal set. The descendants of $Conn(A, y)$ are the subset $\{Conn(A, y'), Conn(y', y)\}$, which contains $Conn(A, y')$. Thus, with g_i as the i^{th} goal set in the inference path and c_i as the first conjunct in each goal set, the inference path satisfies the definition for a recursive path.

The definition of recursive inference given above actually covers a much broader class of problems than we have considered so far. For example, the definition includes recursive paths where there are intermediate descendants in between those descendants with repeating conjuncts. The definition also includes paths where the repeating conjunct may have its variables bound before it is actually processed. Consider the simple axiom

$$y = x + 1 \ \wedge Integer(x) \Longrightarrow Integer(y) \ . \tag{4.6}$$

with the query $Integer(2.5)$. One inference path for this problem is shown in Figure 4.4. When the *Integer* conjuncts are expanded, they are each different, since the variable x is already bound. The subsequence of goals

$$\begin{array}{c} Integer(2.5) \\ Integer(1.5) \\ Integer(.5) \\ \vdots \end{array}$$

does not repeat. However, the subsequence

$$\begin{array}{l} 2.5 = x + 1 \ \wedge Integer(x) \\ 1.5 = x + 1 \ \wedge Integer(x) \\ .5 = x + 1 \ \wedge Integer(x) \\ \vdots \end{array}$$

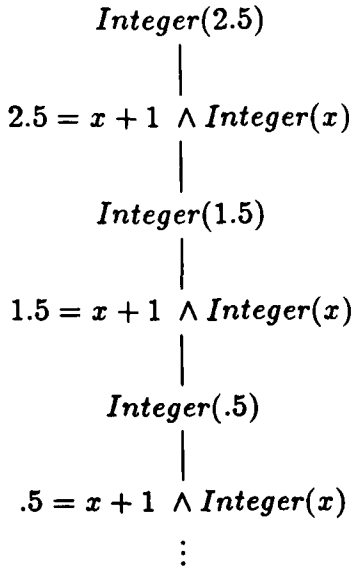


Figure 4.4: Inference path for the query $Integer(2.5)$.

satisfies our definition for recursive inference. Each member contains the conjunct $Integer(x)$, which is both an instance and a descendant of the preceding $Integer(x)$ conjunct.

Although both the integer example and the connectivity example constitute recursive inference, there is an important difference between the two examples. Consider the sequence made up of the conjuncts actually processed by the inference engine at each step. For the connectivity example this sequence repeats.

$$\begin{array}{c} Conn(A, z) \\ Conn(A, y) \\ Conn(A, y') \\ \vdots \end{array}$$

In other words, the repeating conjuncts $Conn(A, \phi)$ are instances of their predecessors at the time they are actually reduced to subgoals. We refer to such recursive inference as *repeating inference*.

In contrast, the sequence of conjuncts

$$\begin{array}{c} Integer(2.5) \\ Integer(1.5) \\ Integer(.5) \\ \vdots \end{array}$$

does not repeat. The argument of $Integer(x)$ is always bound before the conjunct is actually reduced to subgoals, and each time the argument is bound to a different constant.[3] We refer to this non-repeating recursive inference as *divergent* inference.

4.1.4 The Approach

Control of recursive inference means eliminating those portions of the search space that are *superfluous* or *redundant*. (We defined these terms in Chapter 2.) The difficulty is to determine which branches of a search space are indeed superfluous or redundant. If all recursive inference were unproductive it would be a simple matter to provide effective control. However, as we illustrated with some of the examples in Section 4.1.1 there are many instances where a limited amount of recursive inference is necessary in order to arrive at desired answers. If too much of a recursive space is discarded, important answers to the problem are lost. Alternatively, if not enough of the recursive space is discarded, valuable problem solving effort is wasted.

In general, it is not decidable whether or not a given portion of a recursive search space is redundant. However, there are special cases where it is possible to prove redundancy without completely exploring the space. For repeating inference, a simple syntactic solution is possible. We can decide when to cut off inference by keeping track of the answers produced with each additional level of repetition. For divergent inference the problem is much harder. Here we must generate automatic proofs that no answers exist in a portion of the search space. These proofs are similar to proofs of program termination using well-founded sets. They require information about the properties of the relations involved, and about the content of the system's database. Finally, where rule sets are commutative and each set alone cannot produce answers, it is possible to generate automatic proofs that no novel answers will appear in a portion of the search space, again by making use of knowledge about the properties of the relations involved, and about the contents of the system's database.

4.1.5 Organization

In the next section we consider the types of facts that make recursive inference possible, and consider the conditions under which recursive inference will actually occur. The reader more interested in a solution to the problem of recursive inference can skip ahead to Sections 4.3 and 4.4 and refer back as needed to Section 4.2. In Section 4.3, techniques for the common special case of *repeating inference* are

[3]The latter qualification is important. It is still possible for the sequence of goals processed to be repeating even though all of the arguments are bound. We will show examples of this in Section 4.3.

reviewed. Although several of the algorithms presented are not novel, we consider them from the viewpoint of search control, introduced in Section 4.1.4. We provide a proof that the methods are correct and consider the conditions under which the pruning strategies are optimal. In addition, powerful methods for dealing with the special cases of transitivity and logical subsumption are described.

The more general class of non-repeating recursive inference is considered in Section 4.4. Here we show how properties of the relations involved and knowledge about the contents of the system's database can be used to demonstrate that a portion of the search space is redundant. Finally, in Section 4.5 we consider the problem of detecting recursive inference so that control can be instituted only when necessary. Related work is also discussed.

4.2 The Conditions for Recursive Inference

4.2.1 Cyclic and Recursive Collections

Suppose that a set of facts can be arranged in the form,

$$\begin{array}{lllll} F_1: & L_1 & \wedge \phi_1 & \Longrightarrow L'_2 \\ F_2: & L_2 & \wedge \phi_2 & \Longrightarrow L'_3 \\ & & \vdots & \\ F_{n-1}: & L_{n-1} & \wedge \phi_{n-1} & \Longrightarrow L'_n \\ F_n: & L_n & \wedge \phi_n & \Longrightarrow L'_{n+1} \end{array}$$

where the consequent, L'_{i+1} of each fact F_i unifies with the premise L_{i+1} in its successor F_{i+1}, and the consequent, L'_{n+1}, of the final rule F_n unifies with the premise L_1 in the first rule F_1. We say that such a set of rules forms a *cycle* and constitutes a *cyclic collection.*[4] For example the set of rules,

$$\begin{array}{l} P(x) \Longrightarrow Q(x) \\ Q(B) \Longrightarrow P(A) \end{array}$$

form a cycle, since $Q(x)$ unifies with $Q(B)$ and $P(x)$ unifies with $P(A)$. A rule can be involved in more than one cycle, so we also refer to the union of any two cyclic collections that share rules as a cyclic collection.

If a common set of bindings is possible for all of the unifications in a cycle, the facts are said to be *recursive* and constitute a *recursive collection.* In other words, a

[4]This notation and terminology is derived from Minker and Nicolas [MN83]. Minker and Nicolas express these definitions in terms of facts in conjunctive normal form. For simplicity we have expressed these definitions in terms of rules.

group of facts is recursive if there is some common set of bindings b for the variables in each of the facts F_1 through F_n such that $L_i|_b = L'_i|_b$ and $L'_{n+1}|_b = L_1|_b$. The cyclic collection given above is not a recursive collection because x cannot be bound to both A and B simultaneously. However, the cyclic collection

$$P(x) \Longrightarrow Q(x)$$
$$Q(y) \Longrightarrow P(y)$$

is a recursive collection since the binding $x : y$ unifies $Q(x)$ with $Q(y)$ and $P(x)$ with $P(y)$. Likewise, the transitivity and symmetry rules, the albino rule, and the Fibonacci rule given in the previous section are all recursive collections.

As with cyclic collections, it is possible for a single rule to be part of more than one recursive collection. We therefore refer to the union of any two recursive collections that share rules as a recursive collection.

4.2.2 Recursive Search Spaces

We say that a search space is *recursive* if it contains a recursive inference path (as defined in Section 4.1.3). It should come as no surprise that recursive collections give rise to recursive search spaces.

Theorem 4.2 *For any recursive collection of facts there is at least one goal that will result in a recursive search space.*

Proof *By the definition of a recursive collection, there is some common set of bindings b for the variables in each of the facts F_1 through F_n such that each $L_i|_b = L'_i|_b$ and $L'_{n+1}|_b = L_1|_b$. From the goal proposition $L_1|_b$, using the rules $R_1, ..., R_n$ it is possible to regenerate the subgoal $L_1|_b$. This process can be repeated arbitrarily many times, resulting in an arbitrarily long inference path.* □

As an example, consider the transitivity rule for circuit connections. The search space in Figure 4.1 shows that the goal $Conn(A, z)$ has a recursive space since each subgoal in the sequence

$$\langle Conn(A, z), Conn(A, y), Conn(A, y'), Conn(A, y''), \ldots \rangle$$

is an instance of the preceding goal.

Corollary 4.3 *If a goal proposition g results in a recursive search space for a given recursive collection, any generalization g' of the goal will also result in a recursive search space.*

By a generalization of a proposition g, we mean a proposition g' such that $g'|_b = g$ for some set of bindings b. For example, since the query $Conn(A, z)$ has a recursive search space, the query $Conn(x, z)$ will also have a recursive search space.

It is natural to ask whether recursive collections are the only kinds of facts that can lead to infinite search spaces. Infinite search spaces can always occur if there is an infinite database, but barring this possibility, the answer appears to be yes.

Conjecture 4.4 *If an infinite path exists in the search space there must be a recursive collection of facts involved in the generation of that path.*

In fact, we believe that a stronger statement holds.

Conjecture 4.5 *A set of n axioms that is not a recursive collection can generate an inference path of at most length $(2^n - 2)a + 1$ where a is the maximum arity (number of arguments) of all relations in the collection.*[5]

Lewis [Lew75] has proven a weaker theorem, but we are not aware of any proof of these conjectures.

4.2.3 Recursive Inference

As we stated in Section 4.1.3, recursive inference occurs when an inference procedure follows a recursive path in a recursive search space. By this definition a recursive search space is a necessary condition for recursive inference, but it is not a sufficient condition. Thus, even though a given problem may have a recursive search space, recursive inference will not necessarily result. The inference procedure must also happen on to a recursive path. Consider the reformulated version of the transitivity axiom for circuit connections (Section 4.1.2). Although the search space for the goal $Conn(A, z)$ is still a recursive space, if the $IConn$ conjunct is always solved first, recursive inference will not result for this goal and the database of connections given.

In general, whether or not recursive inference occurs depends upon

- the characteristics of the recursive collections involved,
- the facts available in the database,
- the search strategy employed by the inference procedure,

[5] We arrived at the formula $(2^n - 2)a + 1$ by empirical generalization of a set of examples, beginning with the cyclic collection

$$P(y, x) \Longrightarrow Q(x, y)$$
$$Q(A, x) \Longrightarrow P(B, x)$$

and progressing to higher arity, more rules and rules involving functional expressions.

- the strategy for evaluating embedded functional expressions (whether they are evaluated or treated syntactically),
- the number of answers desired for the problem, and
- the number of answers actually available for the problem.

The first of these criteria, the characteristics of the recursive collection, influences the shape of the search space and therefore affects the chances of recursive inference. The facts in the database can also affect the likelihood of recursive inference, as we saw with the reformulated version of the transitivity axiom for connections (4.3). In that example, the presence of the facts $IConn(A, B)$ and $IConn(B, A)$, cause a loop, but either fact alone will not.

The search strategy also affects whether or not recursive inference will occur. If non-recursive subgoals are preferred to recursive subgoals, the chances of recursive inference will be less. This is because the inference procedure might be able to find enough answers without ever exploring a recursive portion of the space. Likewise, if non-recursive clauses are preferred to recursive clauses in conjunctive subgoals, the chances of recursive inference will be less. This is because the non-recursive clauses may fail, stopping recursion.

Finally, recursive inference becomes more likely as the ratio of number of solutions sought to number of solutions available increases. If more answers are required, a larger percentage of the space must be searched, making it more likely that recursive paths will be explored.

Unfortunately, there is no simple precise characterization of when recursive inference will or will not occur. Any such characterization would require a classification of all the different possibilities for each factor, and a multi-dimensional table to consider all of the different combinations. When a recursive collection is present, there is always the potential for recursive inference, although, as we have seen, it can sometimes be avoided by careful search.

4.3 Repeating Inference

As indicated in Section 4.1.3, repeating inference is when the sequence of processed goal conjuncts actually repeats. More precisely, there is some infinite subsequence of the goal conjuncts processed, such that each successive conjunct is an instance of its predecessor. Most of the examples that we considered in Section 4.1 were of repeating inference. In particular, the connectivity example had this characteristic, since a goal expression of the form $Conn(A, z)$ is generated and expanded repeatedly in the leftmost branch of the AND/OR search tree.

In repeating inference, a portion of the AND/OR search space is repeated over and over again. To control the search we must determine the level at which the repetition can be cut off. The search space below the cutoff point must not hold any new answers.

4.3.1 Finding a Single Answer

First consider the special case where only a single answer is needed for a query. In such cases, if an answer cannot be found without exploring a repeating portion of the space, no answer can be found at all. As a result, the search space can be pruned drastically.

Theorem 4.6 *If only a single answer is needed for a goal g, any subgoal g' that is an instance of g can be discarded (along with the entire subspace descending from g'). Furthermore, it is* optimal *to do so, in the sense that the simplest proof of an answer for g will not involve a repeating subgoal g'.*

Some defin?sitions are needed to prove this result. An answer for a goal expression g consists of a set of variable bindings, i.e. a substitution list, such that the steps in the search space, when reversed, would constitutes a proof of the goal expression with those bindings substituted in. We refer to that portion of the search space as a proof tree for that particular answer. We will use the operator $\circ$ to refer to the composition of two binding lists, e.g. $\{x : z, y : B\} \circ \{z : A, w : C\} = \{x : A, y : B, w : C\}$. We say that one proof tree is *isomorphic* to another if they are identical up to variable bindings for the goals and subgoals. Note that if t is a proof tree for a goal $g|_b$ there is an isomorphic proof tree for the generalization g, constructable using the same axioms and rules of inference.

Proof *Let t be a proof tree for the goal g that contains a repeating subgoal g'. Let g'' be the deepest repeating subgoal in the proof tree, and let t'' be the sub- proof tree for g''. Since g'' is the deepest repeating subgoal, t'' contains no repeating subgoals. Since $g'' = g|_b$, t'' is also a proof tree for $g|_b$. So there is also a proof tree t^* for g that is isomorphic to t''. Since there is a non-recursive way of finding an answer to g, g' can be discarded. Furthermore, since t^* is isomorphic to a proper subtree of t, t will never be the simplest proof. Thus, the simplest proof will never involve repeating subgoals.* □

Note that in the statement and proof of this theorem we assumed nothing about the relative cost or simplicity of proofs, except that if one proof is isomorphic to a subproof of another, it is simpler.

There is a useful corollary of the above theorem.

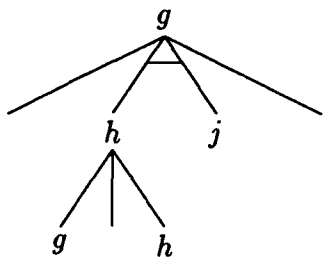


Figure 4.5: Search space for a single-solution problem

Corollary 4.7 *Repeated ground queries and functional queries can always be pruned from a search space.*

This result holds because functional queries never have more than one answer.

Finally, note that Theorem 4.6 does not mean that all repetitions can be discarded, only those for goals that require only one solution. Consider the hypothetical search space in Figure 4.5. The goal g, which has only a single solution, generates a conjunctive descendant $h \wedge j$. It might be necessary to search through several of the answers to the conjunct h in order to find a solution to the conjunction. Thus, while any reoccurences of g can be discarded, reoccurences of the goal h cannot be.

4.3.2 Finding Multiple Answers

In cases where more than one answer is needed, Theorem 4.6 does not apply. Such problems arise far more often than might be expected. Even though only a single answer is needed for a problem, some of its subproblems may be conjunctive, as in the example above. Solving a conjunction frequently requires generating more than one solution to at least one of the conjuncts.

The Theory

If multiple answers are needed, in order to eliminate a portion of the repeating space we must show that that portion of the space is *redundant* (i.e. will not produce any novel answers to the original problem). What makes such a proof possible is the observation that if a search of one or more levels of repetition deeper in a recursive space does not produce any new answers, no amount of additional search will produce any new answers to the original repeated supergoal.

Some notation is needed in order to state this theorem precisely and prove that it is correct. Let $S(g)$ refer to the search space beginning with the goal g and

containing all of the legal descendants of the goal g. A *frontier set* F of a search space $S(g)$ is defined to be a set of goals in the space such that no goal in the frontier set is a descendant of any other goal in the frontier set. Intuitively, a frontier set is some possibly jagged, partial slice through a search space. Let $S_F(g)$ refer to that portion of the space $S(g)$ not including any of the frontier goals $f \in F$ or their descendants $S(f)$. In other words, the restricted search space $S_F(g)$ is just $S(g)$ with all of the frontier branches pruned out. Let $A_F(g)$ refer to the set of answers to the goal g present in the restricted space $S_F(g)$.

For a recursive space let $R_n(g)$ refer to the frontier set consisting of the n^{th} level repetitions of the goal g.

Theorem 4.8 *Let F be the frontier set $R_n(g)$ consisting of n^{th} level instances of the goal g. Let F' be a frontier set consisting of repeating descendants of goals in the set F. If $A_{F'}(g) = A_F(g)$, all of the frontier subspaces $S(r)$ for $r \in F$ are redundant.*[6]

Proof *The proof is by induction on the level of repetition in the search space. First we prove the theorem for the case where $F' = R_{n+1}(g)$.*

Let g' be a first level repeating descendant of g and let c be the set of bindings such that $g' = g|_c$. Let r be the subset of $R_n(g)$ that are descendants of g' as shown in Figure 4.6. Thus $r = R_{n-1}(g')$. Let r' be the set $R_1(r) = R_n(g')$ (all first level repeating descendants of r) and let r'' be the set $R_2(r) = R_{n+1}(g')$ (all second level repeating descendants of r).

The space $S_{r'}(g')$ is an instance of a portion of the space $S_F(g)$. In fact,

$$A_{r'}(g') = \{a : c \circ a \in A_F(g)\} \ .$$

Likewise the space $S_{r''}(g')$ is an instance of a portion of the space $S_{F'}(g)$ so

$$A_{r''}(g') = \{a : c \circ a \in A_{F'}(g)\} \ .$$

Since $A_F(g) = A_{F'}(g)$, it follows that $A_{r'}(g') = A_{r''}(g')$, i.e. there are no additional answers to g' available by going a level deeper.

Let $b_{g'}$ be the set of bindings connecting answers to the subgoal $g' \in R_1(g)$ to answers to the supergoal g. In other words, if a' is an answer to g', $b_{g'} \circ a$ is an answer to g. Then,

$$A(g) = A_{g'}(g) \cup \{a : a = b_{g'} \circ a' \wedge a' \in A(g')\} \ .$$

[6]This theorem relies on the assumption of complete indexing in the problem solver's database. In other words, the system must be able to find any fact in the database that matches a goal. Without complete indexing, answers could be found to an instance of a goal when they could not be found for the original goal. A weaker version of the theorem still holds if complete indexing of the problem solver's database is not assumed. In this case, the frontier set F must contain repetitions instead of instances of the goal g. Essentially, this means that search must be a few recursion levels deeper until a specialization of the initial goal is found for which F will contain pure repetitions.

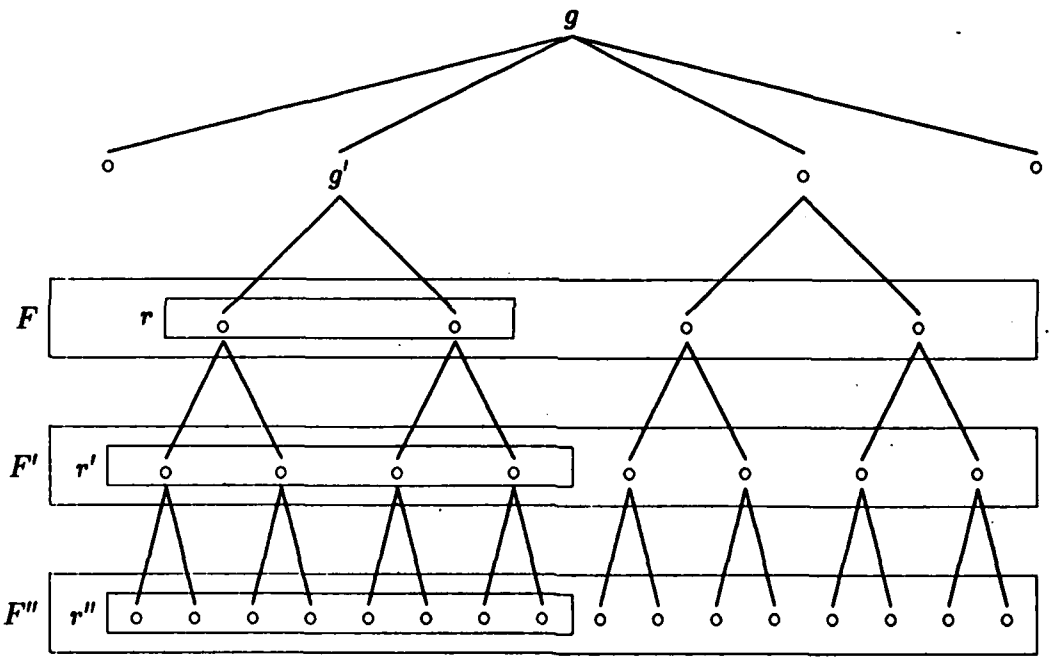


Figure 4.6: Abstract repeating search space.

Now consider the frontier $F'' = R_{n+2}(g)$. *Using the two results above,*

$$\begin{aligned} A_{F''}(g) &= A_{R_1(g)}(g) \cup \bigcup_{g' \in R_1(g)} \{a : a = b_{g'} \circ a' \wedge a' \in A_{r''}(g')\} \\ &= A_{R_1(g)}(g) \cup \bigcup_{g' \in R_1(g)} \{a : a = b_{g'} \circ a' \wedge a' \in A_{r'}(g')\} \\ &= A_{F'}(g) \, . \end{aligned}$$

By induction $A_{F^{(k)}}(g) = A_F(g)$ *for all* k. *Thus,* $A(g) = A_F(g)$, *which means that the repeating descendants in* F *are redundant.*

Finally, for any set F' *satisfying the requirements of the theorem,* $S_{R_{n+1}(g)}(g) \subseteq S_{F'}(g)$, *so* $A_{F'}(g) = A_F(g)$ *implies that* $A_{R_{n+1}(g)}(g) = A_F(g)$. *Since the theorem holds for* $F' = R_{n+1}(g)$ *it holds for arbitrary* F'. □

Corollary 4.9 *The depth of repetition in a search space can be limited to one less than the total number of answers desired for the problem.*

Theorem 4.6 is a special case of this corollary.

Example: Consider the connectivity axiom for circuits,

$$Conn(x, y) \wedge Conn(y, z) \Longrightarrow Conn(x, z) \, .$$

As before, suppose that the problem is to find all points in a circuit connected to a given point A,

$$\textit{find all } z\textit{: } Conn(A, z) \, .$$

An initial portion of the backward AND/OR search space for this problem is reproduced in Figure 4.7. If there are no answers in the system's database for $Conn(A, z)$, there are no answers at all. In this case the frontier sets $F = \{\text{“}Conn(A, z)\text{”}\}$ and $F' = \{\text{“}Conn(A, y)\text{”}\}$ satisfy the conditions of Theorem 4.8. $S_F(g)$ is the null space and $S_{F'}(g)$ is the space consisting of only the goal $g = \text{“}Conn(A, z)\text{”}$. Since there are no answers in the database for the goal g, $A_F(g) = A_{F'}(g) = \emptyset$. As a result, Theorem 4.8 states that no search is necessary for the problem.

If, instead, the database contains the fact $Conn(A, B)$, but no facts about the connections to B, the sets

$$F = \{\text{“}Conn(A, y)\text{”}\}$$

and

$$F' = \{\text{“}Conn(A, y')\text{”}\}$$

satisfy the theorem. In this case, $S_F(g)$ and $S_{F'}(g)$ both contain the single answer $z = B$. As a result, only database answers to the initial goal $Conn(A, z)$ need be located in this case.

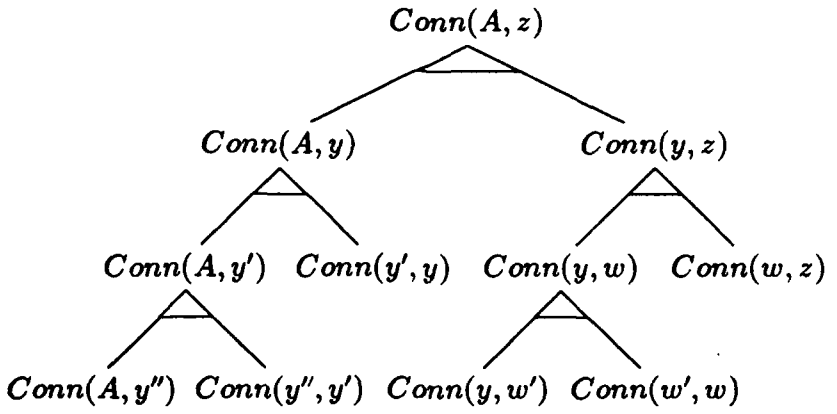


Figure 4.7: A portion of the backward search space for the goal $Conn(A, z)$

Finally, suppose that the database contains the facts $Conn(A, B)$ and $Conn(B, C)$ but no other connections to A, B or C. For this case, the cutoff frontiers contain two terms since the right hand branch of the conjunction also contributes a recursive branch for the binding $z = B$.

$$F = \{\text{“}Conn(A, y')\text{”}, \text{“}Conn(B, w)\text{”}\}$$

$$F' = \{\text{“}Conn(A, y'')\text{”}, \text{“}Conn(B, w')\text{”}\}$$

For both of these frontiers, the answer set will consist of $z = B$ and $z = C$.

Optimality Although Theorem 4.8 tells us some conditions under which a portion of the search space is redundant, it does not tell us that pruning the redundant portion of the space is necessarily the best thing to do. In some cases it can be advantageous to search part of the redundant portion of the space. As an example, consider the connectivity example of the previous section, where the available facts were $Conn(A, B)$ and $Conn(B, C)$. Suppose that we also have the (non-recursive) collection of facts

$$H(x, y) \Longrightarrow Conn(x, y)$$
$$G(x, y) \Longrightarrow H(x, y)$$
$$F(x, y) \Longrightarrow G(x, y)$$
$$E(x, y) \Longrightarrow F(x, y)$$

together with the facts $E(A, B)$ and $E(A, C)$. In this case, we could find all the answers to the query $Conn(A, z)$ by exploring this non-recursive path. Theorem 4.8, therefore, allows us to conclude that the goal $Conn(A, y)$ is redundant. However, the non-recursive way of finding the answer $z = C$ is longer and probably more

costly than finding the same answer by exploring a level deeper in the repeating space. As a result, pruning the subgoal $Conn(A, z)$ would not be the best thing to do in this case.

In the case where all of the solutions are needed to a problem, we can show that pruning the redundant portion is a good idea.

Theorem 4.10 *For recursive problems where all of the solutions are sought, if there exists a frontier F that obeys the conditions of Theorem 4.8 it is* optimal *to prune the frontier goals F, in the sense that the amount of search required to find all answers can only be reduced by this pruning.*

Proof *In order to find all answers in a space, all portions that may contain novel answers must be searched. Assuming that we do not know which portions of $S_F(r)$ are redundant with $S(r)$ for each $r \in F$ then, $S_F(g)$ must be searched in any case. If $S_F(g)$ must be searched, each of the $S(r)$ contain only redundant answers, so there is no advantage to searching any of them. As a result, the amount of search necessary to find all answers can only be reduced by pruning the goals in F.* □

As we demonstrated in the example above, this result does not hold for problems where some specific number of solutions is sought.

Repetition Cutoff Algorithms[7]

In order to make use of Theorem 4.8 we need a mechanism for finding the repetition level that satisfies the conditions of the theorem. Finding such a frontier set requires preserving the answers to any goal with repeating descendants.

Algorithm 4.11

1. If a goal g_i is an instance of one of its supergoals g, the goal g_i is suspended until all other alternatives for solving g have been exhausted.

2. If any new answers are found to the goal g, all repeated instances g_i of g are enabled for another level of expansion. If not, the inference is terminated.

A flowchart of a problem solver incorporating this procedure appears in Figure 4.8. One major efficiency improvement can be made on this procedure. The answers produced by expanding the search space an additional level of repetition will be a subset of those produced in the first level since each repeated descendant g_i is an instance of the goal g. Therefore, it is not necessary to reproduce the space at each

[7]These algorithms were first discovered by Black [Bla68] and were later rediscovered by McKay and Shapiro [MS81] and by the authors.

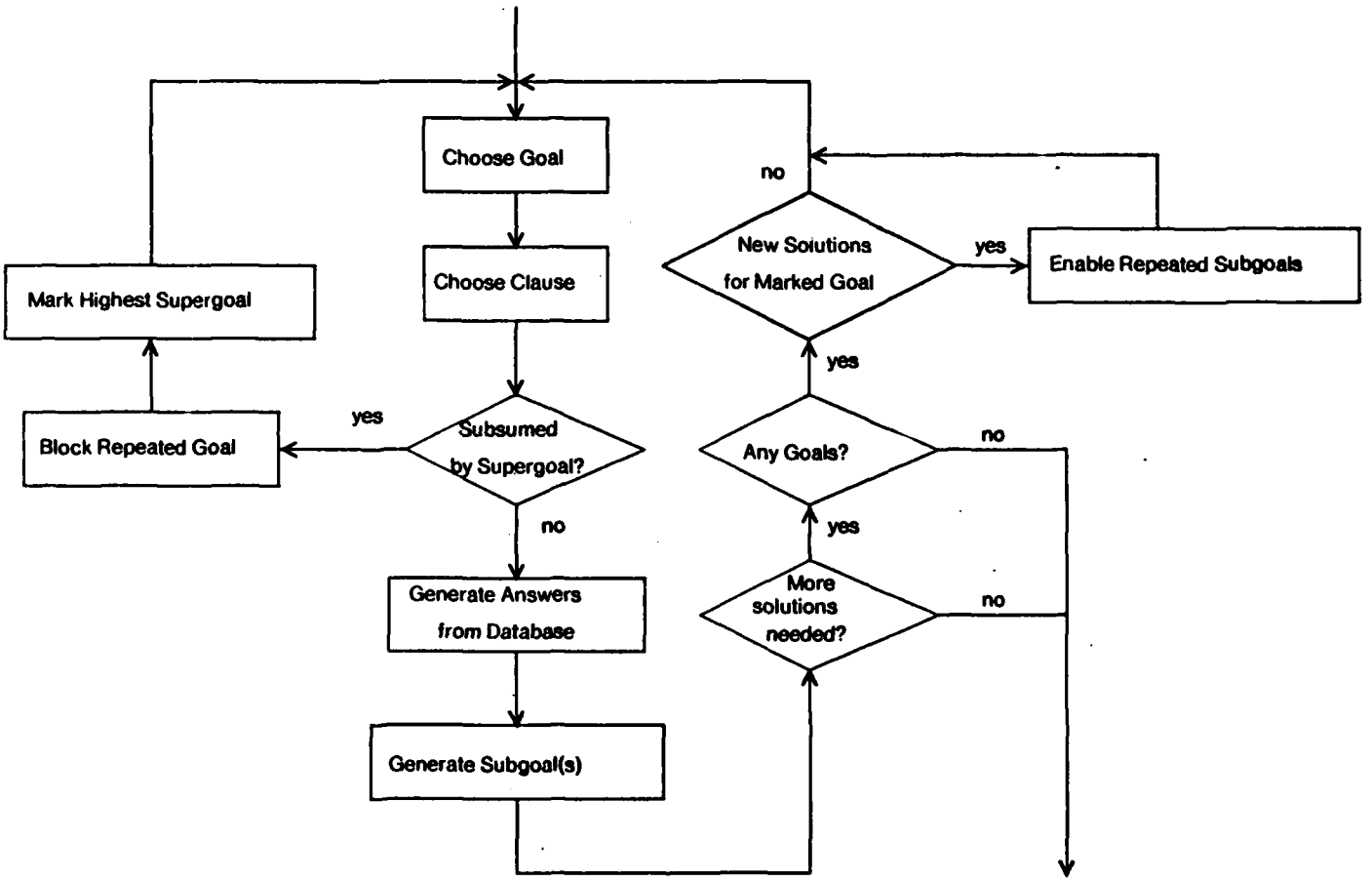


Figure 4.8: Backward inference procedure with repetition control.

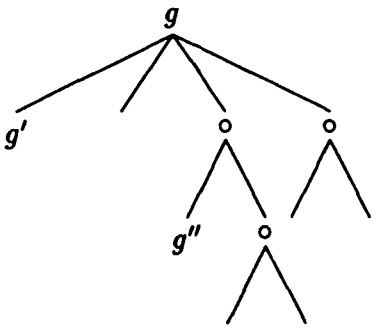


Figure 4.9: An idealized AND/OR tree containing repetition

level. It is sufficient to cache all of the answers to the supergoal and use them as the answers to any repeated descendants. Thus, a more efficient procedure would be

Algorithm 4.12

1. Each time a solution is found to a query (or subquery) the solution is cached.
2. When a repeated descendant is encountered, only instances found in the system's database (including cached answers) are used as solutions to the repeated descendant. No additional inference is performed on this repeated descendant.
3. The solution of a repeated descendant is not complete until no additional solutions can be found to the goal that it is a repeat of. In other words, new answers to a goal must continually be plugged into all repeated descendants until quiescence occurs and no new answers appear.

As an illustration of this method, consider the search tree shown in Figure 4.9. This tree is a snapshot of the goal stack for the inference engine at some point in the computation. There are two repetitions of the original goal expression g, both of which are suspended awaiting answers to g. If the answer α is found to g, this answer is cached and consequently plugged in as an answer for g' and g''. If these branches generate additional answers α' and α'' to g, then these answers, in turn, are cached and must be tried in the two repeated descendants. When no new answers to g can be produced the process is complete.

Example: Circuit Connections Consider the connectivity example of Section 4.3.2 where the goal is to find all points in a circuit connected to a given point A

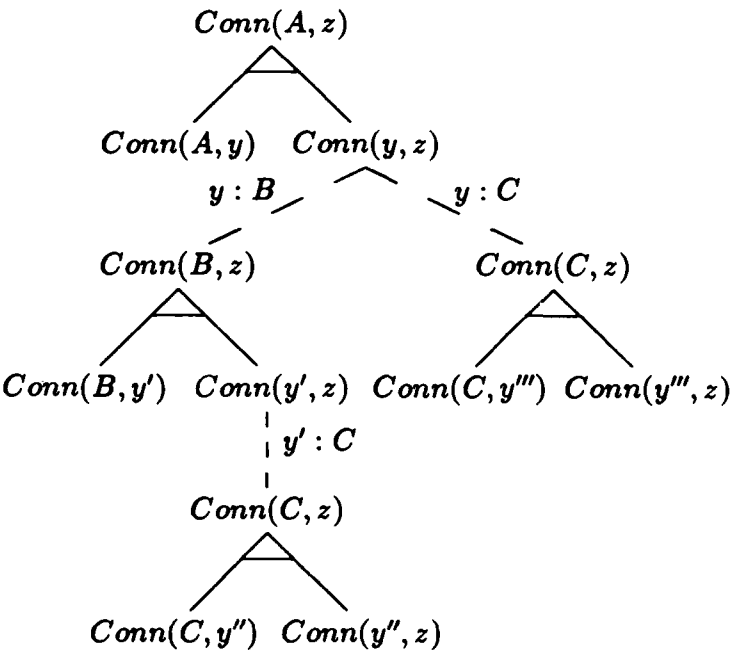


Figure 4.10: Search for the query $Conn(A, z)$

and the database contains the facts $Conn(A, B)$ and $Conn(B, C)$. First the answer $z : B$ is found. The transitivity rule is then applied to the initial goal yielding $Conn(A, y) \wedge Conn(y, z)$. Since the clause $Conn(A, y)$ is an instance of the initial goal, $Conn(A, z)$, no inference is performed on this clause. However, since there is already a cached solution to the original goal $Conn(A, z)$ the solution $y : B$ is found for the repeated descendant. Substituting this binding into the other conjunct yields the subgoal $Conn(B, z)$. The answer $z : C$ is found in the system's database and is therefore cached as a solution to the original goal. The descendant $Conn(B, z)$ is then expanded using the transitivity rule, yielding the conjunction $Conn(B, y') \wedge Conn(y', z)$. As with the first expansion, the clause $Conn(B, y')$ is an instance of the subgoal $Conn(B, z)$ so no inference is performed on the repeated clause $Conn(B, y')$. As before there is already a solution, $y' : C$, in the system's database, so this solution is substituted into the other conjunct yielding the subgoal $Conn(C, z)$. There are no solutions to this clause in the system's database. The expansion to $Conn(C, y'') \wedge Conn(y'', z)$ again contains the repeating subgoal $Conn(C, y'')$, so no further inference is performed and no answers are found to $Conn(C, z)$. This leaves no further alternatives for the supergoal $Conn(B, z)$. However, this subgoal did generate an additional answer $(z : C)$ to the initial goal $Conn(A, z)$, so the cached fact must be used in the first repeating descendant $Conn(A, y')$. Substituting the binding $y' : C$ into the other conjunct yields the subgoal $Conn(C, z)$. As before, this subgoal yields no solutions, and the inference process terminates.

Example: Ancestry As a second example, consider the problem of finding all albinos, given the rule

$$Albino(x) \wedge Parents(z) = \{x, y\} \wedge Albino(y) \Longrightarrow Albino(z)$$

Assume that our database contains the facts:

$$Parents(ABCD) = \{AB, CD\}$$
$$Parents(AB) = \{A, B\}$$
$$Parents(CD) = \{C, D\}$$

$$Albino(A)$$
$$Albino(B)$$
$$Albino(C)$$
$$Albino(D)$$

Beginning with the conjunct $Albino(z)$ the system would first discover the four answers in its database. It would then apply the recursive rule resulting in the conjunction $Albino(x) \wedge Parents(z) = \{x, y\} \wedge Albino(y)$. The first of these conjuncts is identical to its parent so the algorithm would halt further inference on

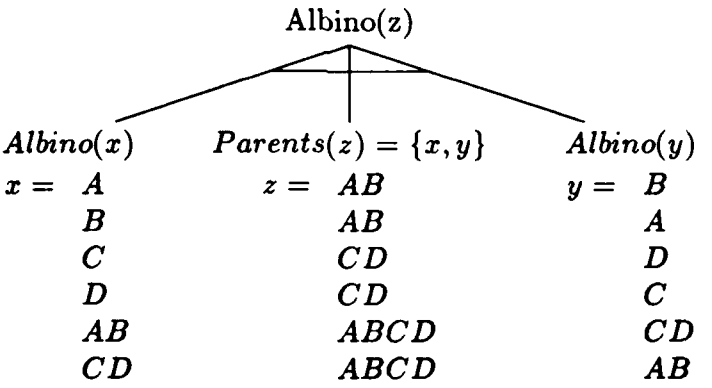


Figure 4.11: Search space for the query $Albino(z)$.

this branch. However, since there are already four cached solutions to the original problem, these are substituted in as solutions to the repeated descendant. We are left with the conjunction $Parents(z) = \{x, y\} \wedge Albino(y)$ for the cases of $x = A$, $x = B$, $x = C$ and $x = D$. For these different bindings, the parents conjunct yields values for y and z:

x	y	z
A	B	AB
B	A	AB
C	D	CD
D	C	CD

For each of these solutions for y, the final conjunct $Albino(y)$ is verified by reference to the database. Thus, the answers AB and CD are produced and cached for the original query. These, in turn, are substituted into the repeated descendant, again yielding the conjunction $Parents(z) = \{x, y\} \wedge Albino(y)$ for the cases $x = AB$ and $x = CD$. The parents conjuncts yield new values for y and z:

x	y	z
AB	CD	$ABCD$
CD	AB	$ABCD$

Again the conjuncts $Albino(CD)$ and $Albino(AB)$ are verified by reference to the database so the answer $ABCD$ is produced and cached for the original query.

Finally, substitution of $ABCD$ into the repeated descendant yields no additional answers (since $ABCD$ has no progeny) and the search terminates. A sketch of this search space appears in Figure 4.11.

Soundness, Completeness and Optimality

Since Algorithms 4.11 and 4.12 are merely ways of finding frontier sets that satisfy Theorem 4.8, they do not adversely affect the logical soundness or completeness of an inference procedure. Although use of these algorithms will result in a drastic reduction of the size of repeating search spaces, their use does not guarantee termination of search. This is because the restricted search space $S_R(g)$ can still be infinite. Theorem 4.8 and the algorithms do not detect or eliminate divergent inference paths. As a result, an inference procedure making use of the cutoff algorithms might still encounter a divergent path and might therefore never terminate or find all of the answers in the space.

However, barring divergent paths, inference procedures based on Algorithms 4.11 and 4.12 are guaranteed to terminate, and any solution in the search space will be found. If the subgoal generator is logically complete, such an inference procedure will also be logically complete.

The algorithms limit search to one recursion level beyond the minimal level that satisfies the conditions of Theorem 4.8. In Section 4.3.2 we pointed out that cutting off redundant recursive paths at the earliest possible level may not be optimal. This observation therefore extends to the algorithms as well.

4.3.3 Special Types of Repetition

The theorems of the previous section are general, but weak. There are special cases of repeating inference where stronger results are possible. For example, in Section 4.3.1 we developed a much stronger result for the case where only a single answer was needed. There are two other special cases that merit particular attention, descendant subsumption, and transitivity. Both of these cases rely on *goal subsumption* for their power. We say that a goal set g *subsumes* another goal set g' if there is a set of bindings b such that $g|_b \subseteq g'$. If only a single answer is needed for a problem any goal set g' subsumed by another goal set g will be redundant with that descendant [Nil80]. The situation is somewhat more complicated when more than one answer is needed.

Theorem 4.13 *Let h' and h'' be arbitrary descendants of g and let b' and b'' be the binding sets that relate solutions to h' and h'' to solutions to g (i.e. $h' \Longrightarrow g|_{b'}$ and $h'' \Longrightarrow g|_{b''}$) as illustrated in figure 4.12. Suppose that h' subsumes h'' with the bindings c. If the bindings for the output variables in $b' \circ c$ are a subset of the bindings b'', h'' is redundant with h'.*

Proof *Let a'' be a solution found to h''. Then $b'' \circ a''$ is a solution to g that will be found by exploring h''. We must show that the same solution (or a generalization of it) can be found by exploring h'. Since $h'|_c \subseteq h''$ we know that there is some solution $a' \subseteq c \circ a''$ that will be found for h' (assuming complete database indexing). Thus,*

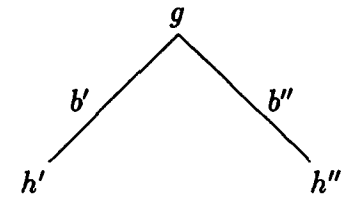


Figure 4.12: Illustration for the subsumption theorem

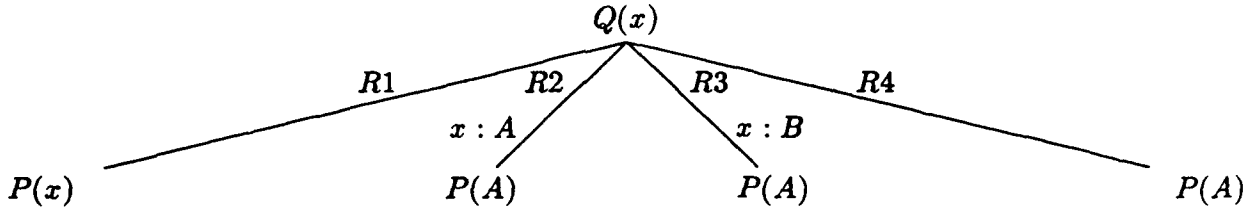


Figure 4.13: Subsumption example

$b' \circ a'$ will also be a solution to g. But since $a' \subseteq c \circ a''$ and $b' \circ c \subseteq b''$ we have $b' \circ a' \subseteq b' \circ c \circ a'' \subseteq b'' \circ a''$. So the solution $b' \circ a'$ found by the descendant h' is a generalization of the solution $b'' \circ a''$ found by the descendant h''. h'' is therefore redundant with h'. □

As an example, consider the simple search space of Figure 4.13 generated from the goal $Q(x)$ using the rules

$$\begin{aligned} R1:&\ P(x) \Longrightarrow Q(x) \\ R2:&\ P(A) \Longrightarrow Q(A) \\ R3:&\ P(A) \Longrightarrow Q(B) \\ R4:&\ P(A) \Longrightarrow Q(x)\,. \end{aligned}$$

The subgoal $P(x)$ subsumes the three subgoals $P(A)$ with the bindings $c = \{x : A\}$, but not all of these subgoals are redundant with $P(x)$. The leftmost $P(A)$ in Figure 4.13 is redundant because its binding set $\{x : A\}$ is identical to c. However, the second $P(A)$ has the binding set $\{x : B\}$, which does not contain $x : A$. Therefore, this subgoal is not redundant with the subgoal $P(x)$, unless x is not an output variable. The third instance of $P(A)$ is also not redundant with the subgoal $P(x)$ (assuming x is an output variable), since it has an empty binding set. Finally, it is worth noting that the first and second instances of $P(A)$ are redundant with the third instance of $P(A)$. For these cases c is empty and therefore, is contained in any binding set.

Descendant Subsumption

We can apply Theorem 4.13 to cases of repeating inference. In this case, h'' is a descendant of h', and the root goal g can be taken to be h'. Thus, the binding list b' is empty.

Corollary 4.14 *Let g' be a repeating descendant of g and let b be the set of bindings that relate solutions to g' to solutions to g ($g' \Longrightarrow g|_b$). Let c be the set of bindings such that $g' = g|_c$. If the bindings for the output variables in c are contained in b, the goal g' is redundant with the goal g. Furthermore, it is* optimal *to discard g', since for every proof of an answer to g involving g', there is a simpler proof of the same answer not involving g'.*

As an example, consider the simple search space of Figure 4.14 generated from the goal $P(x)$ and the rules

$$R1: \; P(A) \Longrightarrow P(A)$$
$$R2: \; P(A) \Longrightarrow P(B)$$
$$R3: \; P(A) \Longrightarrow P(x)\,.$$

The three subgoals, $P(A)$, are subsumed by the root goal $P(x)$ with the binding

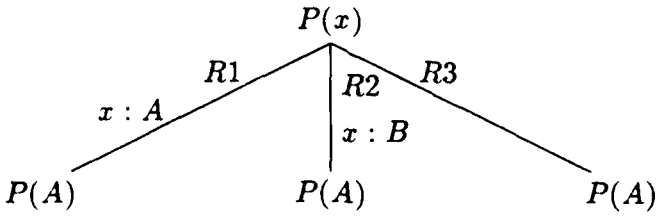


Figure 4.14: Repetition subsumption example

set $c = \{x : A\}$. The first subgoal has the binding set $b = \{x : A\}$, which is identical to the binding set c. According to Corollary 4.14 the first subgoal can therefore be eliminated. This agrees with our intuitions, since any proof of $P(A)$ could be used directly to get the solution $x : A$ for the original goal $P(x)$. The second subgoal has the set of bindings $b = \{x : B\}$, which does not contain c. Therefore, it cannot be eliminated if x is an output variable. Intuitively, a proof of the second subgoal, $P(A)$, would allow us to conclude $P(B)$, and there may be no direct proof of $P(B)$. Likewise, the third subgoal cannot be eliminated since its binding set b is empty and therefore does not contain c. In this case, a proof of $P(A)$ would allow us to conclude $\forall x P(x)$, and there may be no direct proof of this statement.

The most common cases of descendant subsumption are when a descendant is identical to an ancestor in every respect, including variables. For this case, the

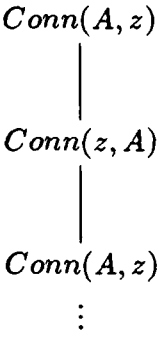


Figure 4.15: Search path for a symmetry rule.

set c is empty and the descendant can be eliminated. These situations arise from if-and-only-if rules expressing definitions and from rules expressing properties like symmetry, associativity, and commutivity. For example, in a circuit analysis system we might need the information that electrical connections are symmetric.

$$Conn(x, y) \Longrightarrow Conn(y, x)$$

Suppose we were to apply this rule to the problem of finding all the points in a circuit connected to a given point A.

$$\textit{find all } z\textit{: } Conn(A, z)$$

We first get the subgoal $Conn(z, A)$. Applying the rule to this subgoal gives the subgoal $Conn(A, z)$, as shown in Figure 4.15. But this subgoal is identical to the original goal, so we can prune the repetition using the subsumption theorem.

Transitivity

The subsumption theorem also has a direct application to repeating inference resulting from transitivity rules. Consider the connectivity example used in the previous sections, with the query $Conn(A, z)$ and a database containing the facts

$$Conn(A, B)$$
$$Conn(B, C)$$
$$Conn(C, D)\ .$$

Figure 4.16 shows the portion of the space that would be generated for this problem using Algorithm 4.12. First, the answer $z = B$ is found in the database. Then, the conjunctive subgoal $Conn(A, y) \wedge Conn(y, z)$ is generated. The first of these conjuncts is repeated, so we plug in the answer $x = B$ that has already been

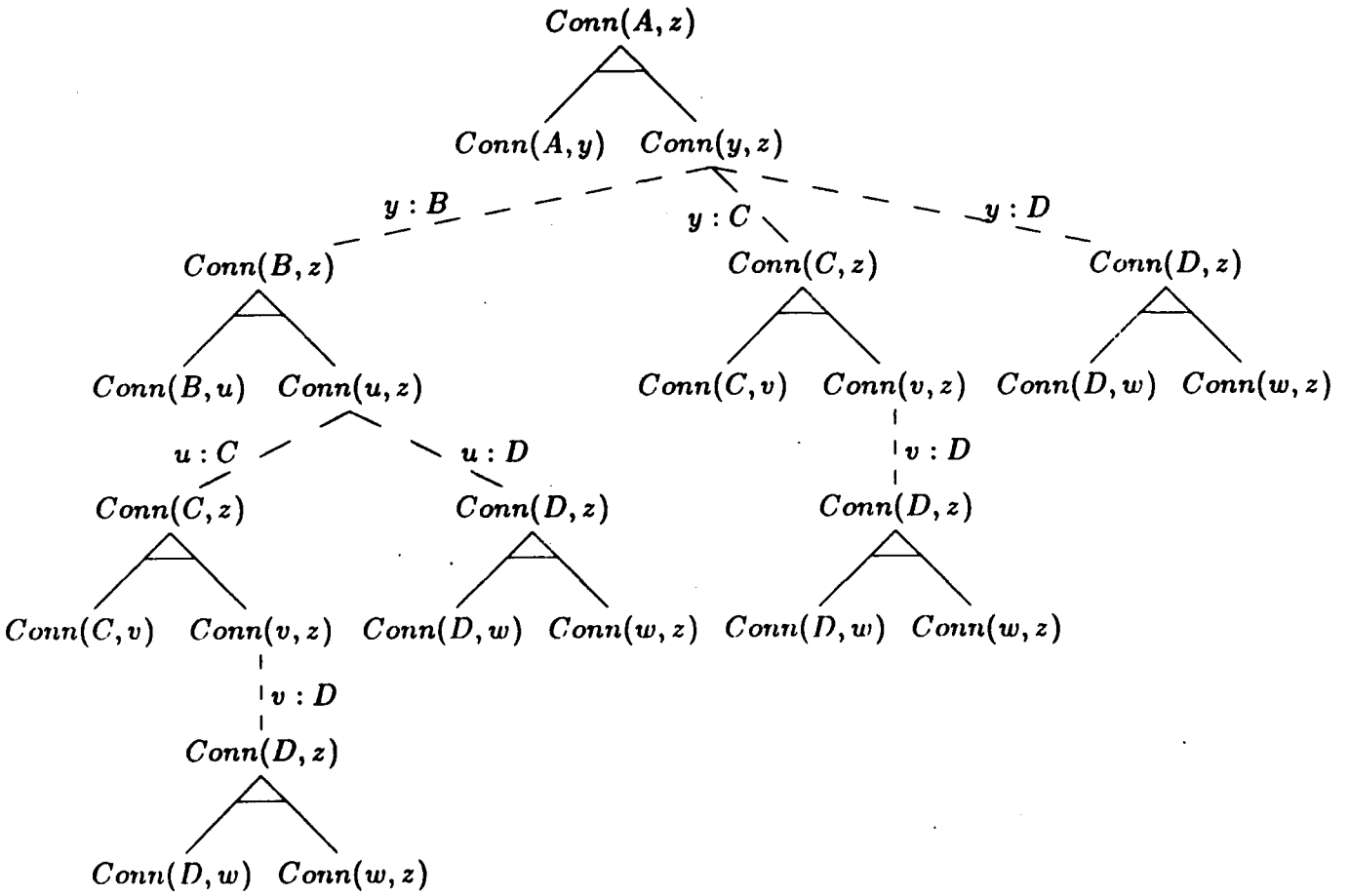


Figure 4.16: Search space for the query $Conn(A, z)$.

found. Continuing on the remaining conjunct $Conn(B, z)$ we find one answer in the database, $z = C$, and use the transitivity rule to generate the subgoal $Conn(B, u) \wedge Conn(u, z)$. The first of these conjuncts is a repeat of its parent so we again plug in the solution already found, $u = C$. The remaining conjunct becomes $Conn(C, z)$. Again, there is a single answer in the database, $z = D$. We apply the transitivity rule one more time yielding the conjunction $Conn(C, v) \wedge Conn(v, z)$. The first of these is again a repeat of its parent and we plug in the available solution, $v = D$. The remaining conjunct, $Conn(D, z)$, yields no solutions, so we begin to unwind. Note that we have already found all of the solutions to the problem, $z = B, C$ and D, yet we have not substituted the newly generated answers into the two remaining repeating descendants. According to the algorithm, we must substitute $u = D$ into the conjunction $Conn(B, u) \wedge Conn(u, z)$. Following this, we must substitute the answers $y = C$ and $y = D$ into the first subgoal $Conn(A, y) \wedge Conn(y, z)$. Each of these substitutions causes more redundant inference. In effect the procedure produces each of the answers twice. A similar situation occurs with the dual query $Conn(x, D)$ (assuming the conjuncts are processed in a reasonable order) and with the general query, $Conn(x, z)$.

Much of the duplication can be eliminated by recognizing and pruning subsumed goals. For example, the two instances of the goal $Conn(C, z)$ are mutually redundant according to the subsumption theorem. Likewise, the four instances of the goal $Conn(D, z)$ are mutually redundant. If all but one of each are eliminated the remaining search space does not contain any redundant portions. Using the subsumption theorem, together with Algorithm 4.12 therefore solves the problem. However, the two results can be combined into a more succinct reduction theorem.

Theorem 4.15 *Let $g' \wedge g''$ be the conjunctive subgoal produced by applying a transitivity rule*

$$R(x, y) \wedge R(y, z) \Longrightarrow R(x, z)$$

to the goal g, as illustrated in Figure 4.17. Let h' and h'' be the conjunctive subgoals produced by the application of the transitivity rule to the conjuncts g' and g'' respectively. Then, h' and h'' are mutually redundant. In other words, $S(g) = S_{h'}(g) = S_{h''}(g)$.

Proof *For all possible g that match $R(x, z)$, the conjunction $g' \wedge h''$ subsumes $h' \wedge g''$ and vice versa. For example, if $g =$ "$R(x, z)$", $g' =$ "$R(x, y)$", $g'' =$ "$R(y, z)$", $h' =$ "$R(x, v) \wedge R(v, y)$" and $h'' =$ "$R(y, w) \wedge R(w, z)$", then $R(x, y) \wedge R(y, w) \wedge R(w, z)$ subsumes $R(x, v) \wedge R(v, y) \wedge R(y, z)$ and vice versa for any subset of $\{x, z\}$ as output variables. The theorem therefore follows immediately from the subsumption theorem.* □

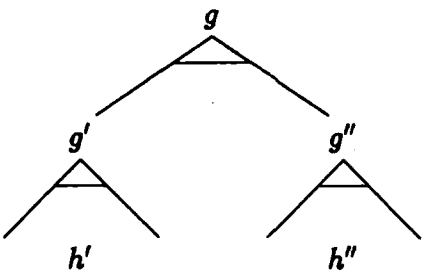


Figure 4.17: Transitivity search space.

This result is easily implemented. When a transitivity rule is applied to a goal, the rule should not be applied to one of the two conjunctive subgoals generated. For the connectivity example the two possibilities are shown in Figures 4.18 and 4.19. If the transitivity rule is not reapplied to left-hand branches the result is a simple, but lopsided search space. If it is not reapplied to the right hand branch, repeating inference occurs in the left-hand branch, and the methods of Section 4.3.2 must be applied. Using Algorithm 4.12, inference on the left-hand branch would stop after one level. All answers are generated merely by caching solutions and substituting them into the left branch.

4.4 Divergent Inference

The most troublesome form of inference loops are those that do not repeat. Consider again the simple rule describing the integers:

$$y = x + 1 \ \wedge \mathit{Integer}(x) \Longrightarrow \mathit{Integer}(y) \tag{4.7}$$

A query such as $\mathit{Integer}(2.5)$ generates an infinite sequence of subgoals like that shown in Figure 4.20. If we were to list the sequence of goal conjuncts reduced at each step in the inference process, no specific conjunct would appear more than once in this sequence. There are an infinite number of *Integer* conjuncts in this sequence, but each one has a different argument. As we indicated in Section 4.1.3 we refer to such non-repeating recursive inference as *divergent* inference.

How do we go about cutting off inference in such cases? In general it is only semi-decidable whether or not the space below a given subgoal will contain novel answers to the problem. Yet for a case like the one above we can supply a fairly simple argument for pruning the infinite recursion from the search space. Suppose that the smallest integer in the database is 2. The sequence of descendants from $\mathit{Integer}(2.5)$ is monotonically decreasing. As a result, once we have passed

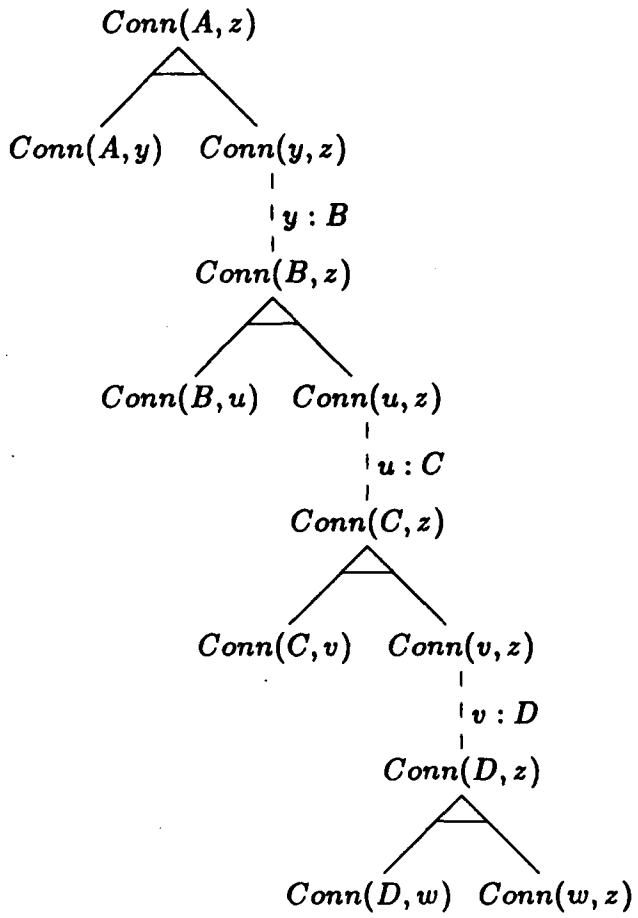


Figure 4.18: Left-pruned search space for the goal $Conn(A, z)$

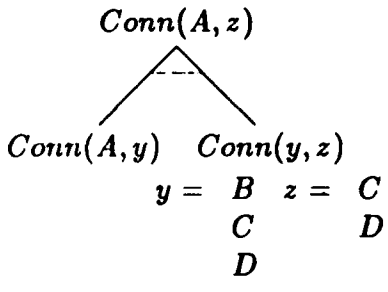


Figure 4.19: Right-pruned search space for the goal $Conn(A, z)$.

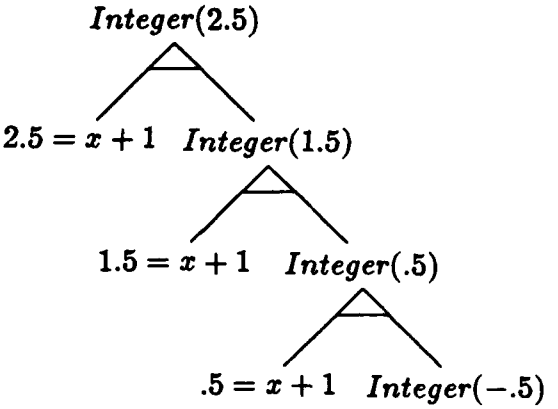


Figure 4.20: Search space for the query *Integer*(2.5). The first conjunct has been evaluated for each subgoal.

Integer(2) all further descendants are superfluous; they will never be able to match any fact in the database. This argument is not unlike the sort of arguments used in proving program correctness or program termination. Here we have used, as an invariant assertion, the fact that every descendant of *Integer*(2.5) will be of the form $\phi(x) \wedge Integer(x)$ and that x will always be less than 2.

This kind of argument can be generalized to arbitrary recursive collections. What is necessary is to find an invariant assertion for each goal form in the loop that implies that there will be no answers in the database for the corresponding goal. In addition we must show that all other rules that apply to goals in the loop (rules not in the recursive collection) will not produce any answers.

We can make this kind of argument more precise. Let the relation $No(p)$ mean that there are no facts in the database that unify with the proposition p.

Theorem 4.16 *Let* $\{R_1, \ldots R_m\}$ *be the relations occurring in the consequents of the rules in a cyclic collection (recursive collections included). Let*

$$F_{j,k,n} = \text{“}\phi_{j,k,n}(\vec{y}, \vec{z}) \wedge R_j(\vec{y}) \Longrightarrow R_k(\vec{z})\text{”}$$

designate the n^{th} *rule in the collection having a relation* R_j *in its premise, and* R_k *in its consequent. (The* $\phi_{j,k,n}$ *may contain additional* R *from the set.) Suppose that there is a predicate* β_k *on the domain of each relation* R_k *such that*

1. $\beta_k(\vec{y}) \Longrightarrow No(\text{“}R_k(\vec{y})\text{”})$,
2. $\beta_k(\vec{z}) \wedge \phi_{j,k,n}(\vec{y}, \vec{z}) \Longrightarrow \beta_j(\vec{y})$, *and*
3. $\beta_k(\vec{z}) \Longrightarrow Superfluous(\gamma)$, *for all other facts,* $(\gamma \Longrightarrow R_k(\vec{z}))$, *not in the recursive collection.*

Then, if $\beta_k(\vec{A})$ holds, the goal $R_k(\vec{A})$ is superfluous.

Here $\beta_k(\vec{z})$ is the invariant assertion for those goals with the relation R_k. The first condition states that the invariant assertion assures that no answers will be found. The second condition states that the invariant assertions are preserved from a goal to its immediate subgoals, and the third condition assures that none of the exit points of the loop will lead to any answers.

Proof *First we consider just those descendants of $R_k(\vec{A})$ that can be generated using rules in the cyclic collection. We want to show that there are no answers in the database for any of these descendants. We know that each of these descendants will contain a clause $R_j(\vec{y})$ for some R_j in the set of relations described in the theorem. If we can show that the invariant assertion $\beta_j(\vec{y})$ holds for that descendant, condition (1) in the theorem tells us that there will not be any answers in the database. Thus, we want to show that, for each descendant g' generated using only the cyclic collection, there is some expression $\psi(\vec{y},\vec{z})$ and some relation R_j in the set so that g' takes the form*

$$g' = \text{“}\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})\text{”} \tag{4.8}$$

and that

$$\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})\ . \tag{4.9}$$

We prove this by induction on descendant depth. For the initial goal $R_k(\vec{A})$, the induction hypothesis (4.8) holds if we let ψ be the empty clause, $\vec{y} = \vec{A}$ and let $j = k$. Likewise, (4.9) follows from the given, $\beta_k(\vec{A})$.

Now assume (4.8) and (4.9) for every l^{th} level descendant of the goal $R_k(\vec{A})$. Any $(l+1)^{st}$ level descendant will be a subgoal of some l^{th} level descendant. There are two possible ways of obtaining subgoals from an l^{th} level descendant $\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})$.

1. *Apply some rule to a clause of ψ. In this case the new subgoal will be of the form $\psi'(\vec{y},\vec{z}) \wedge R_j(\vec{y})$, which satisfies (4.8). Furthermore, since $\psi'(\vec{y},\vec{z}) \Longrightarrow \psi(\vec{y},\vec{z})$ and $\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$ we get $\psi'(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$, which proves the second half of our induction hypothesis (4.9).*
2. *Alternatively, we could apply some rule $F_{i,j,n}$ to the clause $R_j(\vec{y})$. In this case the new subgoal will be $\psi(\vec{y},\vec{z}) \wedge \phi_{i,j,n}(\vec{x},\vec{y}) \wedge R_i(\vec{x})$. If we let $\psi'(\vec{x},\vec{z}) = \phi_{i,j,n}(\vec{x},\vec{y}) \wedge \psi(\vec{y},\vec{z})$ our subgoal becomes $\psi'(\vec{x},\vec{z}) \wedge R_i(\vec{x})$, which again satisfies (4.8). Furthermore, since $\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$ and $\beta_j(\vec{y}) \wedge \phi_{i,j,n}(\vec{x},\vec{y}) \Longrightarrow \beta_i(\vec{x})$ (condition (2) of the theorem) we get that $\phi_{i,j,n}(\vec{x},\vec{y}) \wedge \psi(\vec{y},\vec{z}) \Longrightarrow \beta_i(\vec{x})$, or $\psi'(\vec{x},\vec{z}) \Longrightarrow \beta_i(\vec{x})$. Thus the second part of the induction hypothesis also holds.*

The hypotheses (4.8) and (4.9) therefore hold for all $(l+1)^{st}$ level descendants of $R_k(\vec{A})$ and by induction, for all descendants of $R_k(\vec{A})$ produced using only rules

in the cyclic collection. It follows from condition (1) in the theorem that there will not be any answers in the database for any of these descendants.

What remains is to consider those descendants produced using rules not in the cyclic collection. Every such descendant will involve either applying such a rule directly to the goal $R_k(\vec{A})$, or to one of the goals $\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})$ generated using the cyclic collection. Again there are two ways of producing subgoals to a goal of the form $\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})$.

1. *Apply some rule to a clause of ψ. As before, such a subgoal will still satisfy the induction hypothesis and the previous argument holds.*
2. *Apply a rule, $\gamma \Longrightarrow R_j(\vec{y})$ to the clause $R_j(\vec{y})$ to yield the subgoal $\psi(\vec{y},\vec{z}) \wedge \gamma$. But by the induction hypothesis we know that $\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$. By condition (3) of the theorem we conclude that γ is superfluous. Thus there are no answers in the database to any of the descendants of this subgoal.*

Therefore, there are no answers in the database for any of the descendants of $R_k(\vec{z})$, which means it is superfluous. □

4.4.1 Example

To see how this theorem applies, consider the simple integer example introduced earlier. For this example there is only one rule in the recursive collection. The relation in its consequent (R_1) is the *Integer* relation, and its ϕ will be $\phi_{1,1,1}(x,y) =$ "$y = x + 1$". If we choose $\beta_1(x) =$ "$x < 2$", the condition $\beta_1(y) \wedge \phi_{1,1,1}(x,y) \Longrightarrow \beta_1(x)$ will be satisfied. If the smallest integer in the database is 2, $\beta_1(x) \Longrightarrow No(R_1(x))$ is true. The final condition, that all rules not in the recursive collection result in superfluous goals, is true since there are no other rules. Then, according to the theorem $\beta_1(x) \Longrightarrow Superfluous(\text{"}R_1(x)\text{"})$, or $x < 2 \Longrightarrow Superfluous(\text{"}Integer(x)\text{"})$. Therefore, we conclude that the subgoal $Integer(1.5)$ is superfluous.

4.4.2 Application of the Theorem

In general, mechanizing the application of Theorem 4.16 is not a simple matter. First we choose an applicable recursive collection to apply the theorem to. It may be that all recursive collections are already known and marked in the database. In this case finding an applicable recursive collection is a straightforward lookup operation. If not, we must recursively enumerate the set of applicable rules looking for recursive collections. This is done by mapping through each rule that applies to a goal and doing the same for each subgoal. If the same rule is used again in any path, a cycle and possible recursive collection has been located. If several independent recursive collections are found we must choose one. In satisfying the

final criterion of the theorem (that all other applicable rules do not result in any answers) the others will be considered. Note that the theorem may need to be applied recursively to prove these cases.

Ambiguity in choosing the recursive collection also arises when two or more recursive collections share a common rule. In this case we have nested or interwoven loops. According to the definition of a recursive collection, their union also constitutes a recursive collection. We could therefore choose to apply the theorem to one of the individual recursive collections, or to the composite collection. If we choose to apply it only to an individual collection, in the final step it will be necessary to prove that none of the other interwoven loops can yield an answer. This is usually more difficult. It is therefore probably wise to consider the maximal recursive collection first.

The second step is to collect the set of consequent relations in the recursive collection. This is straightforward.

The third step is to find a set of invariants β_i that satisfy the characteristics

1. $\beta_k(\vec{z}) \Longrightarrow No(\text{“}R_k(\vec{z})\text{”})$, and
2. $\beta_k(\vec{z}) \wedge \phi_{j,k,n}(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$.

This task involves generating possible predicates β_i and testing them to see if they satisfy the above axioms. The most efficient way to do this is to start at one place in the loop, and proceed around the loop in an orderly fashion, generating the β at each step. Thus we start by generating a possibility β_k for some k and check to see that it satisfies the first axiom above. Then, choose j so that there is some rule $F_{j,k,n}$. Now generate β_j and check to see that it satisfies both the first and second axioms. Then, choose i so that there is some rule $F_{i,j,n}$ and so forth.[8]

The real problem is in generating good candidates for any individual β_i, particularly since the desired β_i might be a conjunction of known relations. We could start by considering all known predicates on the domain D_i of R_i. If none of these work, we could try all known relations from D_i to a new domain D' and conjoin these with known predicates on D'. If none of these work, we consider conjunctions containing three relations, and so on. In general this may be necessary. However, it seems that β often takes the form of an integer bound,

$$\beta_i(\vec{x}) = \text{“}\gamma(\vec{x},l) \wedge l \sim N\text{”}$$

where $\sim$ is one of $<, >$ *or* $=$ and N is a fixed integer. In our simple integer case, γ was the identity relation, $\sim$ was $<$ and N was 2. However, if we were dealing with lists of ever increasing length, the *Length* function might be appropriate. Similarly,

[8]Manna and Waldinger [MW77] discuss more sophisticated ways of generating loop invariants for the purposes of program verification. Much of this work appears to be applicable here.

if we were dealing with human ancestry a function such as *Birthdate* might be appropriate. The strategy for generating β_i therefore involves first considering an empty γ if the domain of R_i is the integers. If this fails, known relations mapping the domain D_i onto the integers are considered. In effect, this is a way of generating possible ordering relations for domains where an ordering relation is not already known. Although it may, in theory, be necessary to consider conjunctions of known relations for γ, if the number of known relations is large, the space of possibilities quickly becomes intractable.

The final step involves verifying that none of the other relevant facts will generate any answers to the problem. This may be trivial as in the integer example, or it can be arbitrarily difficult, if there are other recursive collections involved. In the latter case, this final step may well involve application of Theorem 4.16 or any one of the cutoff theorems developed in Section 4.3.

4.4.3 Functional Embedding: A Special Case

A common cause of divergent inference are rules that contain functional expressions on their left hand sides. By this we mean rules of the form,

$$P(F(x)) \wedge \phi \Longrightarrow P(x) .$$

For example, the rules,

$$Jewish(Mother(x)) \Longrightarrow Jewish(x)$$
$$Integer(Successor(x)) \Longrightarrow Integer(x)$$

are both of this sort. Such rules will always lead to divergent inference if the inference engine does not evaluate the embedded functional expressions. For example, a query such as $Jewish(Job)$ would lead to the subgoals $Jewish(Mother(Job))$, $Jewish(Mother(Mother(Job)))$, etc.

For such cases we can often choose β to be a lower bound on the level of functional embedding in a subgoal. If there are no rules relevant to a problem that can shrink the amount of functional embedding (e.g. $P(f(f(x))) \Longrightarrow Q(X)$), it is possible to stop the inference process whenever the level of functional embedding exceeds the largest embedding available in the database. For example, in the jewish ancestry problem, if the fact with the largest functional embedding is $Jewish(Mother(Mother(Job)))$, any subgoal having a functional embedding deeper than two could be discarded. Notice that this strategy refers to total functional embedding independent of the actual functions involved. The reason is that there may be rules available such as $P(G(x)) \Longrightarrow P(F(x))$ that can result in new subgoals having different embedded functions.

4.4.4 Commutivity of Inference Steps

In the previous section we considered only cases where it was possible to prove that no answers existed in a portion of the search space. In fact we can generalize Theorem 4.16 by only insisting that subgoals contribute no novel answers to the overall problem (as opposed to no answers at all). It is usually quite difficult to prove that answers generated somewhere in a loop will not result in novel answers to the overall goal. However, there are special cases where redundancy can be proven, and in such cases Theorem 4.16 can be applied. In this section we develop such a special case result for situations where the inference steps are *commutative.*

Consider the pair of axioms:

$$R_1: \ y = x + 1 \wedge Integer(x) \Longrightarrow Integer(y)$$
$$R_2: \ y = x - 2 \wedge Integer(x) \Longrightarrow Integer(y)$$

As before, suppose that the problem is to determine whether or not 2.5 is an integer, and the smallest integer in the database is 2.

When only the first of the above two rules was available, we argued that the sequence of subgoals from $Integer(1.5)$ was monotonically decreasing, and therefore the subgoal $Integer(1.5)$ was superfluous. Given both rules, this argument no longer holds, since $Integer(3.5), Integer(5.5), \ldots$ are now descendants of the subgoal $Integer(1.5)$. In fact, imagine that the fact $Integer(5.5)$ happened to be in the data base. Then, the subgoal $Integer(1.5)$ would not be superfluous, since the problem could be solved by exploring one of its descendants.

Even though the goal $Integer(1.5)$ may not be superfluous, we can argue that it is redundant with its supergoal, $Integer(2.5)$. The argument depends on the observation that any application of the two rules above is *commutative.* In other words, if a subgoal can be produced by applying one rule, then the other, it can also be produced by applying the rules in the reverse order. For example, the subgoal $Integer(3.5)$ can be produced from the goal $Integer(2.5)$ by applying R_1 followed by R_2. Alternatively, it can be produced by applying R_2 followed by R_1.

Using this observation, we separate the descendants of $Integer(1.5)$ into two groups, those generated by applying only the first of the two rules, and those that involve at least one application of the second rule. From our earlier argument we know that the first class will not result in any answers. For the second class, since the two rules are commutative, the same subgoal can be produced by first applying the second rule to the goal $Integer(2.5)$. Thus the subgoal $Integer(1.5)$ is redundant.

We now make this kind of argument precise. We say that two sets of rules are commutative if any subgoal produced using a rule from one set followed by a rule from the other set could also be produced by using the rules in the opposite order.

More formally,

$$Commutative(s,t) \Longleftrightarrow \forall q \in s, r \in t, g, g', h$$
$$(Subgoal_q(g,g') \wedge Subgoal_r(g',h)$$
$$\Longrightarrow \exists g'' Subgoal_r(g,g'') \wedge Subgoal_q(g'',h))$$

where the notation $Subgoal_r(g,h)$ means that the goal h can be derived as a subgoal of the goal g using the rule r.

Theorem 4.17 *Suppose that the set of applicable facts for a goal g can be broken up into two commutative subsets s and t. Suppose that all descendants of g, generated using only rules in s, produce no novel answers (i.e. if only rules in s are used, the subgoal g would be redundant). Then, all (immediate) subgoals of g produced using rules in s are redundant for the entire collection $s \cup t$.*

Proof *Let g' be an (immediate) subgoal of g generated using a rule from the set s. Consider an arbitrary descendant, d, of g'. Suppose d is produced using only the rules in the set s. Then, by the premises of the theorem there are no answers in the database for d that result in novel answers to the overall problem. Alternatively, suppose that d is produced using, at least one rule r from the set t. Since the rules in s and t are commutative, the descendant d will also be a descendant of the subgoal g'' generated by applying r to the goal g. All descendants of g' (including g' itself) are therefore redundant with immediate subgoals of g produced using rules in t. Since our choice of g' was arbitrary all immediate subgoals of g produced using rules in s are redundant.* □

4.4.5 Example

Using the theorem above, together with the results of previous sections, the two rule integer example can now be handled.

There are two immediate subgoals of the goal *Integer*(2.5). The first, *Integer*(1.5) is redundant according to Theorems 4.17 and 4.16. Therefore, it is possible to eliminate it without sacrificing any answers. The other subgoal, *Integer*(4.5), has the two immediate subgoals *Integer*(3.5) and *Integer*(6.5). If the maximum integer in the database is 3, we can again apply Theorems 4.17 and 4.16 to show that *Integer*(6.5) is redundant. The other subgoal, *Integer*(3.5) has the two immediate subgoals *Integer*(2.5) and *Integer*(5.5). The latter is again redundant, by application of Theorems 4.17 and 4.16. The remaining subgoal *Integer*(2.5) is identical to the original goal, and so by Theorem 4.14 it can also be eliminated. The abbreviated subgoal tree is shown in Figure 4.21.

It is interesting to note that if we changed the constant in either of the two rules to an irrational number, the inference could not be completely stopped. We could

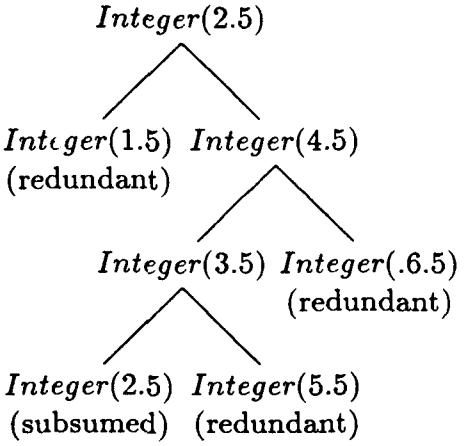


Figure 4.21: Search space for the query *Integer*(2.5). The first conjunct has been evaluated and removed from each subgoal

still apply Theorems 4.17 and 4.16 at each level of the search space, but we would never return to the original goal, and therefore could never apply Theorem 4.14. In this case, the chain of subgoals would continue to bounce back and forth between the two extreme integers in the database.

4.4.6 Remarks

In this section, we first developed a general theorem for terminating divergent inference, and used it to solve a simple problem involving a single loop. We also applied the theorem to the special case of functional embedding.

It is generally more difficult to apply the theorem to cases of redundancy, as we can see from the two rule integer example above. Powerful special case methods like Theorem 4.17 seem essential for dealing with such complex cases. We have investigated only one such result here. Additional work is needed to build up a library of theorems, like 4.17, that can be used for various difficult cases of divergent inference.

4.5 Discussion

4.5.1 Detecting Recursive Inference

There is always a price to pay for controlling inference and control of recursive inference is no exception. For repeating inference the cost is relatively low. It involves suspension of repeating goals and the caching of answers to goals with

repeating subgoals. However, for divergent inference, control involves explicit proofs that subspaces are superfluous or redundant. When the alternative is an infinite loop, any finite control cost is justifiable. The problem is, we usually cannot be certain whether or not an infinite loop will result. As we pointed out in Section 4.2, even when a problem has an infinite recursive search space, recursive inference will not necessarily occur. The necessary answers might be found before an infinite path is explored by the inference engine.

In general, it is undecidable whether or not a given inference procedure will terminate when searching a recursive space.[9] The best that can be done is to institute control only when it is considered likely that it will be necessary or cost-effective. This general issue is discussed further in Section 7. For recursive inference there are several interesting strategies. The simplest is to monitor search depth or total search space size and institute control when it exceeds some threshold. A more elaborate, but more costly scheme is to preserve the subgoal and justification trees and institute control when a given fact has been used more than some fixed number of times in the derivation of a particular subgoal. A third alternative is to limit control to those cases where a recursive collection is involved in the deduction. (Recursive collections could be recognized either when rules are entered into the system, or during the problem solving process.)

Each of the strategies has certain advantages and disadvantages. In general, they trade accuracy for expense. For example, the recognition of rule reuse is usually a more accurate predictor of recursive inference than overall search depth, but it is also more expensive since it requires keeping the goal and justification stacks and searching them for each new subgoal.[10] As a result, the best strategy for a given application will depend upon such things as the average depth of inference, the frequency of recursive inference, and the density of recursive collections in the system's database.

There is also no reason why these strategies cannot be combined. For example, we could use search space depth or complexity to determine whether or not to initiate the strategy of checking for recursive collections or repeated rules. Likewise, the strategy of checking for repeated rules could be used as a filter for the strategy of looking for recursive collections. These combined strategies allow a less expensive but less accurate detection criteria to serve as a filter for a more accurate and more costly one. Such combinations may, in fact, prove to be the most cost-effective for many applications.

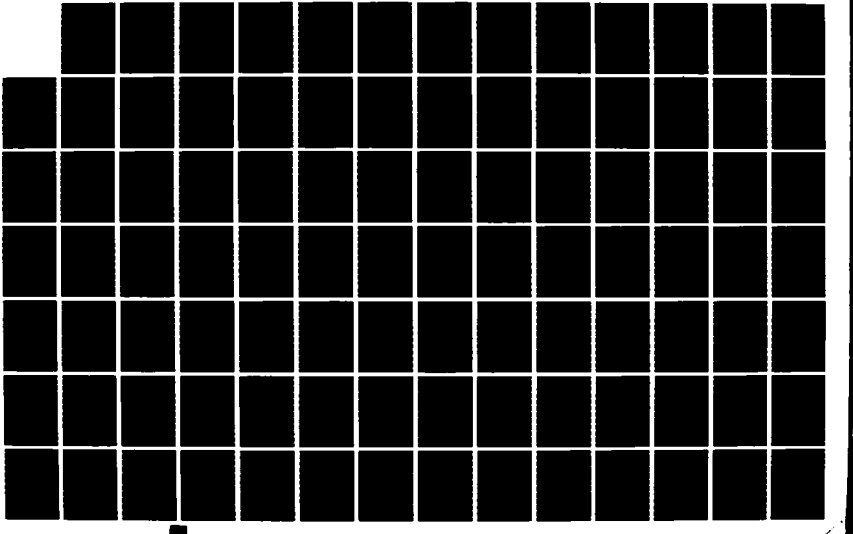
---

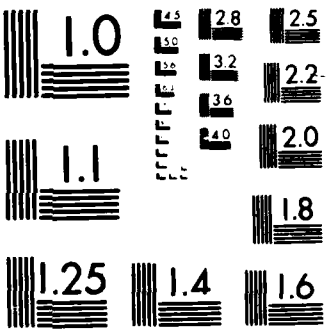
[9]The problem is equivalent to the halting problem for Turing machines since backward inference over a set of axioms is Turing equivalent.

[10]Associating a marker or counter with each rule doesn't work in general. The marker would have to be path dependent since we do not wish to count the repeated usage of rules in independent inference paths.

AD-A172 502 CONTROLLING INFERENCE(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE D E SMITH APR 86 STAN-CS-86-1107 2/3
N00014-81-K-0004

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

4.5.2 History and Related Work

Recursive Inference

Black [Bla68] and McKay and Shapiro [MS81] describe algorithms for stopping repeating inference similar to those developed in Section 4.3.2. However, they do not provide any proof that the pruning strategy is correct and do not consider the question of optimality. They also do not consider any of the special cases (like transitivity, subsumed subgoals, or single answer queries) where more powerful strategies can be used.

The special case of eliminating identical subgoals appears to have been first used by Gelernter in his geometry theorem proving program. [Gel63].

> *Subgoals ... are rejected ... that appear as higher subgoals on the [subgoal] graph (or are syntactically symmetric to some higher subgoal). – page 142*

In Gelernter's application since all goals and subgoals are geometry theorems requiring only a yes or no answer both Theorems 4.6 and 4.14 apply. As a result, all repeated subgoals can be eliminated for this particular application. Loveland and Reddy [LR81] have shown that the technique of eliminating identical subgoals can be extended to backward inference mechanisms that recognize and make use of "contradiction constructs".

Special cases of repeating inference were also used in building the MYCIN system [Sho84]. One cause of repeating subgoals in MYCIN was the use of *self-referencing* rules. A self-referencing rule states that if there is already evidence for a condition ψ and some other condition ϕ holds, there is additional evidence for the condition ψ. These rules therefore include the proposition ψ in both the premise and conclusion. MYCIN handles a self-referencing rule by postponing it until all other rules for concluding ψ have been used. Then, the self-referencing rule is applied exactly once.

This strategy involves pruning all repeated applications of a rule, a much stronger pruning strategy than is indicated by Theorem 4.8. This strategy works for self-referencing rules because they are actually quite different from recursive rules. Consider how we would translate a self-referencing rule into a precise declarative statement. We might be inclined to write something like

$$\psi \wedge \phi \Longrightarrow_p \psi$$

where $\Longrightarrow_p$ means that we have p additional evidence for the conclusion (the actual calculus for combining certainties is unimportant). If this statement were true, given some small amount of evidence for ψ, and the fact that ϕ is true, we could

use this axiom over and over again to derive greater and greater belief in ψ.[11] This is not the intended meaning of the self-referencing rule. In a self-referencing rule the recursive premise is a screen to prevent exploration of ϕ unless there is already some evidence of ψ. In other words, the recursive premise is control knowledge about when to apply the simpler rule

$$\phi \Longrightarrow_p \psi \ .$$

Thus a logical translation of a self-referencing rule would consist of two rules; the simple rule given above, and a control rule indicating that the above rule should not be tried unless there is already evidence for ψ. Neither of these rules are recursive, so the theorems developed in Section 4.3 do not apply. This translation also shows why MYCIN's pruning strategy for self-referencing rules is appropriate. Since the above rule is not recursive, it need be applied only once.

Repeating inference also occurs in MYCIN as a result of rule loops. For example, a rule might allow a parameter B to be inferred from a parameter A, while another rule might allow A to be inferred from B. MYCIN's strategy in such cases is to never use a rule more than once in a single reasoning chain. This is equivalent to pruning all repeating subgoals. This works because, once the context is bound, the premises and conclusions of such rules are ground clauses. Thus, as in Gelernter's application, the powerful pruning strategy of Theorem 4.14 applies.

More recently, Minker and Nicolas [MN83] have developed a special case of Corollary 4.14 and have shown that for the class of "singular" recursive rules all repeating subgoals will be subsumed and can therefore be eliminated.

Other approaches to controlling repeating inference have also received some attention, although the results have been limited to special cases. Reiter [Rei78b] and Minker and Nicolas [MN83] have shown conditions where it is possible to use only forward inference on recursive collections. Automatic reformulation of recursive collections has been explored by Chang [Cha80] and Naqvi and Henschen [NH80,HN84]. They describe methods of automatically generating efficient procedures for the special class of "regular" recursive collections. Minker and Nicolas [MN83] have shown that this method applies to a slightly broader class of recursive collections. Recently, Ullman, Van Gelder, and Naughton [Ull85,Nau86] have obtained results for several special cases of recursive inference. In particular, they have proposed techniques similar to those suggested in Section 4.4.3 for handling the special case of functional embedding. They have also isolated classes of recursive collections where the recursion can be eliminated by reformulating the rules involved.

[11] It is interesting to note that, Theorem 4.6, as stated, will not hold in the case of uncertain reasoning. Even if ψ is a ground clause, recursion could continue to increase the belief in ψ. However, from an evidential point of view we would never want to allow this, since arguments should not be circular.

The more difficult problem of divergent inference has received little attention in the literature. Fischer Black noted the problem in developing his natural deduction system [Bla68] but provides no solution other than depth-limited search.

Problems of recursive inference have also arisen outside of the Artificial Intelligence community. Both repeating and divergent inference are constant obstacles in the construction of PROLOG programs. Users of PROLOG become well versed in manual reformulation of rules to eliminate infinite loops. As we illustrated, this is not always an easy task and the resulting programs can be quite opaque.

Recently, at Stanford we have encountered repeating and divergent inference in the construction of systems for reasoning about digital circuits [Gen85b,Sin85,Kra84]. The techniques described here are being implemented in an experimental version of the MRS system [Rus85].

Program Verification

There is a strong similarity between recursive inference and recursive programs that do not terminate. In essence, an inference procedure together with a goal and recursive collection of facts is a recursive program. Thus, it should come as no surprise that the techniques for deciding whether a given inference path terminates, loops, or diverges, bear a striking resemblance to the method of well-founded sets used to prove program termination [Flo67,Man74].

But the similarities end here. In general, a recursive program that does not terminate is of little use. It must be modified so that it does terminate. Doing this requires some understanding of the intention behind the program. In contrast, a recursive collection of facts has *meaning*, independent of the particular inference engine being used. In this case, the inference engine must be modified so that it will perform the proper deductions. The intent of the inference procedure is known. Thus, the change takes the form of information about how to prune the search space. In short, a program that does not terminate may be incorrect in an arbitrary manner, while an inference engine that loops (for a given goal and recursive collection) is incorrect in a very specific way; it explores too much of the search space.

4.5.3 Summary and Final Remarks

The control of recursive inference involves demonstrating that portions of a search space are either superfluous or redundant. When either of these properties has been demonstrated, the offending portion of the search space can be discarded. Although this will always be logically correct, it may not be optimal in every case.

Proofs of redundancy and superfluity involve knowledge about the contents of

the system's database, and about properties of the relations involved in the inference, such as ordering relations on the domains, monotonicity, boundedness and commutivity. This kind of information is commonly available but has rarely been needed or used in AI systems. In contrast to the general domain-dependent character of the control problem, the special case of repeating inference admits control that is domain-independent. The method of suspending and reenabling repeated subgoals does not depend upon the meaning of the symbols involved. It is not entirely clear why this fortuitous result should hold. It is true however, that in some cases the more general domain-dependent techniques of proof can lead to more severe pruning for repeating inference than is possible with the syntactic method of Section 4.3.

Determining whether or not recursive inference will occur for a given problem is in general undecidable. We have suggested three possible criteria for determining when control of recursive inference should be instituted; inference depth or complexity, repeated rule usage, and the use of recursive collections. Combinations of these approaches also appear promising, although the decision will almost certainly prove dependent upon the mix of problems encountered in any particular application.

Finally, there is no a priori reason why the techniques of proving redundancy and superfluity could not be applied to non-recursive inference. The limiting factor is cost. When an infinite search is avoided, a high cost is justifiable. However, for non-recursive inference the problem would have to be a difficult one for expensive analysis like that of Section 4.4 to be cost-effective. For such cases, complex monitoring strategies like those proposed in Section 4.5.1 would be indispensable.

Chapter 5

Ordering Conjunctive Queries[1]

5.1 Motivation

In a conjunctive problem the goal expression is a conjunction of clauses that share variables. An answer to such a problem must satisfy each of the goal clauses simultaneously. Such problems occur in every sort of intelligent system as well as in many other kinds of programs. For example, any non-trivial PROLOG program contains rules with conjunctive premises, and many database queries are conjunctions. In problem solvers and planners that deal with a wide range of tasks, conjunctive problems are often given to the top level. They also occur as a result of conjunctive rule premises in the knowledge base. In expert systems, this is the primary source of conjunctive problems. For example, in the MYCIN system [BS84], nearly all of the over 400 rule premises are conjunctions of two or more clauses.

Occasionally, with inference, a system can reduce a conjunctive problem to a single conjunct, and can readily enumerate the answers to that conjunct. But more often, no such reduction is possible, either because the system does not have sufficient knowledge to perform such inference, or because there is no single conjunct that is equivalent to the problem statement. In this event there is little alternative but to generate solutions to one of the conjuncts, substitute those solutions into the remaining conjuncts, and try to solve those remaining conjuncts.

The difficulty with a naive generate and test procedure is that random choice of the conjunct to "generate" next may result in gross inefficiency, or may even make the problem impossible to solve. Some conjuncts may have an infinite number of solutions, or may have far too many solutions to enumerate within practical time limitations. For other conjuncts, it may not be possible for a system to enumerate the solutions at all until other conjuncts have been solved. Thus, in some cases the

[1]This chapter is a slightly revised version of an article that appeared in *Artificial Intelligence* [SG85]

choice of conjunct order is merely a matter of efficiency, but for others it determines whether or not a problem solver or planner will be capable of solving the problem at all.

One application where the ordering problem is manifest is the Intelligent Agents Project at Stanford [FGKM80]. The task of an intelligent agent is to serve as an expert interface for dealing with an operating system. An agent must accept requests from a user, make appropriate plans for realizing those requests, and perform and monitor the operating system commands for carrying out those plans. Nearly every interesting request to an intelligent agent is a conjunction. For example, commands to find, format, and print documentation, or commands to move groups of files are all conjunctive. One simple but important class consists of requests for information. In these cases it is possible to ignore the added complexity of inference and planning and treat the request purely as a conjunctive query. As a specific example, suppose that the user of the system wishes to know all of the Scribe files that are larger than 100 pages and belong to a member of his directory group. Formally this is a request to find f such that

$$\begin{aligned} &\mathit{File}(f) \wedge \mathit{Format}(f, \mathit{Scribe}) \wedge \mathit{Size}(f, s) \wedge s > 100 \\ &\quad \wedge \mathit{DirectoryOf}(f, d) \wedge \mathit{DirectoryGroup}(d, g) \\ &\quad \wedge \mathit{DirectoryGroup}(u, g) \wedge \mathit{CurrentDirectory}(u) \end{aligned} \tag{5.1}$$

where $\mathit{File}(f)$ means that f is a file, $\mathit{Format}(f, s)$ means that the file is in format s, $\mathit{DirectoryOf}(f, d)$ means that the file f is in directory d, $\mathit{DirectoryGroup}(d, g)$ means that the directory d is in group g and $\mathit{CurrentDirectory}(u)$ means that u is the directory that the user is currently connected to.

Given this request, a typical generate and test problem solver would enumerate all of the $\approx 10^4$ files in the system, checking each one to see if it satisfied the format, size, and (lastly) directory group requirements. This is definitely undesirable, if not absolutely unacceptable. In contrast, if the conjuncts are enumerated in the reverse order only the relevant directories are searched. While the ordering of the above conjuncts may appear malicious, it could have been worse. Consider what would happen if the conjunct $s > 100$ was first. Almost as bad would be to have the directory conjuncts in such an order that the problem solver would find all possible file/directory pairs ($\approx 10^6$) before pruning those that do not belong to a member of the same user group. Arbitrary conjunct ordering might be practical for a blocks world problem solver, but it is not feasible for solving problems like the intelligent agent's problem.

Parallel hardware does not solve this problem either. For example, suppose we were to attempt to produce the solutions to each conjunct independently (in parallel) and intersect the resulting sets in an appropriate order. For the intelligent agent's problem this would involve enumerating the solutions to an infinite set as

well as to several sets having $\approx 10^4$ members. In contrast, if the conjuncts are optimally ordered, the total size of the search space is $\approx 10^3$. Likewise, it is not practical to try all possible conjunct orderings in parallel. For the above problem this would require 8! processors.

It is important to recognize that the need for conjunct ordering is not merely a byproduct of our formalism, or with our choice of vocabulary for talking about directories and files. It is true that any given conjunctive problem can be made to disappear by appropriate choice of vocabulary. But, no matter how one chooses a set of predicates for stating facts about the world, it will always be necessary to search for objects having characteristics that can only be expressed as a conjunction of the existing predicates.

If conjunctive problems are so pervasive and conjunct ordering is so important, why have most previous AI systems survived without it? In fact they have not. The system builders do the ordering a priori, so that the system itself does not have to. Anyone who has built a backward reasoning system or a non-trivial PROLOG program is painfully aware that clauses in rule premises must be carefully ordered. It is not just expedient to do so, in many cases it is absolutely crucial.[2]

Fortunately, it has been possible to do this ordering ahead of time for most expert systems because each one solves only a very limited class of problems. For example, a medical diagnosis system is always faced with the same problem of finding the diagnosis (and perhaps therapy) for a patient. The patient may change, and the facts of the case change, but the form of the query does not change. In systems like this, the problem can be stated (and built in) a priori, and the generated subproblems can be optimally ordered by judicious ordering of the premise clauses in the rules. In contrast, it is not always possible to find a single ordering that will be effective for systems that must accept a wide variety of goals.

5.1.1 Approach and Assumptions

We will make the basic assumption that adequate improvement in system performance can be obtained by ordering each specific set of conjuncts right before giving it to the generate and test engine. We call this the *static ordering assumption*. Under this assumption the problem solver can be represented as consisting of a conjunct ordering step followed by the generate and test step, as in Figure 5.1. If s is the set of conjuncts given, the conjunct ordering step can be characterized as

[2]For example, consider the rule

$$Mother(x, y) \wedge Sister(y, z) \Longrightarrow Aunt(x, z)$$

when the goal is to find someone's aunts (e.g. find all of the x such that $Aunt(Tillie, x)$). Attacking the premise clauses in the wrong order requires enumerating all person/sister pairs in the universe

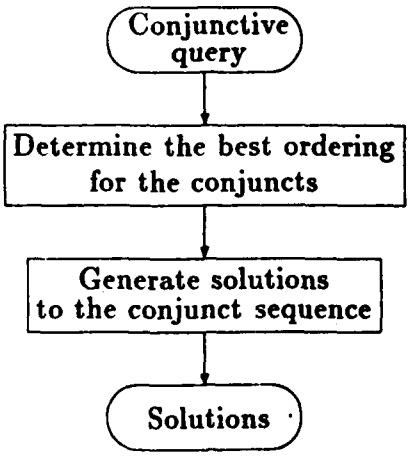


Figure 5.1: A problem solver with conjunct ordering

find t such that $BestOrdering(s,t)$,

where *BestOrdering* is a relation between a set of conjuncts and any optimal ordering for that set. The best ordering for a set of conjuncts is one that will cost the problem solver the least to enumerate

$$BestOrdering(s,t) \iff Ordering(s,t) \wedge \forall x(Ordering(s,x) \implies Cost(t) \leq Cost(x)) . \quad (5.2)$$

The best ordering for the empty set is just the empty sequence and the best ordering for the singleton set is just the singleton sequence

$$BestOrdering(\emptyset, \langle\rangle)$$
$$BestOrdering(\{x\}, \langle x\rangle) .$$

In general, the cost of solving a conjunction depends upon many factors, such as the nature of the problem solving engine, the characteristics of the database available to the problem solver and the characteristics of the problem itself. For the analysis and examples in the body of this paper (Sections 5.2 and 5.3) we have made some limiting assumptions.

First of all, we assume a simple sequential generate and test procedure that enumerates the answers to the next conjunct in the sequence, plugs those answers into the remaining conjuncts and repeats. We assume that it has no special capabilities such as selective backtracking [PP82] or the ability to treat independent conjuncts separately. We will reconsider these assumptions in Section 5.4.1.

Second, we will assume that, for a solvable conjunct, all of the answers are directly available in the problem solver's database. In other words, we assume that no inference is required, or that the cost of the inference is insignificant. In general this is not a reasonable assumption for AI problems. However, it is reasonable for simple information requests of an intelligent agent. We will consider the more general case in Chapter 6.

Third, we assume that functional expressions in a query are eliminated by adding additional conjuncts to the query. Ground functional expressions in the database cause little difficulty, and are therefore allowed, but we assume that no functional expressions containing variables are present in the database.

Finally, we will consider only conjunctive problems that have a finite number of solutions. The reason for this limitation will be explained in Section 5.2 and the assumption will be reconsidered in Section 5.4.3.

5.1.2 Organization

Computing the cost of a given sequence of conjuncts is the subject of Section 5.2. The analysis in this section is detailed, so the reader may wish to skim the material and refer back to the details as necessary in later sections. Following this analysis, in Section 5.3 we consider how to use cost information to order conjuncts effectively. First a general ordering method is presented and analyzed, then several simple conjunct ordering heuristics are examined. We also examine ways of breaking conjunctive problems into independent pieces and show how this characteristic can be used to advantage in the ordering process. Finally we consider the integration of domain specific ordering advice with more general ordering principles.

In Section 5.4 we consider what is required to relax the limiting assumptions introduced in Section 5.1.1 above, and mention some of the open research questions involved in doing so. In particular we consider the effects and prospects of alternative generation strategies and of reordering during the actual problem solving process.

Finally, in Section 5.5, related work is discussed. Within the database community considerable work has been done on the problem of optimizing conjunctive database queries. We discuss differences between the ordering techniques developed in Section 5.3 and those used in database systems. Differences in general methodology are also discussed.

5.2 Cost

5.2.1 The Cost of Solving a Conjunctive Problem

The cost of producing solutions to a sequence of conjuncts depends upon how many solutions are desired. Assuming there are a finite number of solutions to a problem, the cost of enumerating some fraction of those solutions will be that fraction of the cost of enumerating all of the solutions. For example, suppose that a set of conjuncts has ten solutions but only one is needed. On the average, one eleventh of the space must be searched before the first solution is found. The cost will therefore be one eleventh of that required to enumerate all of the solutions. Since cost will be used purely as a comparative measure between two orderings of conjuncts, it suffices to use the cost of computing all of the solutions, regardless of how many solutions are desired.

If $t = \langle p_1, ..., p_M \rangle$ is a sequence of conjuncts, let $Numsol(t)$ refer to the total number of solutions to those conjuncts. If the sequence t is broken into two subsequences $t_{1,i-1}$ and $t_{i,M}$, the number of solutions to t can be expressed as the sum over all solutions to $t_{1,i-1}$ of the number of solutions to $t_{i,M}$ under that particular set of variable bindings.

$$Numsol(t) = \sum_{\{a:Solves(a,t_{1,i-1})\}} Numsol(t_{i,M}|_a) \tag{5.3}$$

If we define the average number of solutions to a sequence t over the set of solutions to another sequence s to be

$$AvgNumsol(t,s) = \underset{\{a:Solves(a,s)\}}{\text{Avg}} (Numsol(t|_a)) \tag{5.4}$$

the number of solutions can be expressed as

$$\begin{aligned} Numsol(t) &= Numsol(t_{1,i-1}) * AvgNumsol(t_{i,M}, t_{1,i-1}) \\ &= \prod_{i=1}^{M} AvgNumsol(p_i, t_{1,i-1}) \,. \end{aligned} \tag{5.5}$$

Assuming that the average number of solutions can be calculated for each individual conjunct, this equation provides a means of calculating the total number of solutions to a problem.

Similarly, let $Cost(t)$ refer to the cost of producing the solutions to the sequence t for a given problem solver. For each of the solutions to the conjunct sequence $t_{1,i-1}$, the variable bindings must be substituted into the remaining conjuncts $t_{i,M}$, and the solutions found for that sequence. The cost for doing this will be the sum of

the cost of producing the solutions to $t_{1,i-1}$ and the cost of producing the solutions to $t_{i,M}$ for each set of solutions to $t_{1,i-1}$

$$Cost(t) = Cost(t_{1,i-1}) + \sum_{\{a:Solves(a,t_{1,i-1})\}} Cost(t_{i,M}|_a) \,. \tag{5.6}$$

Defining the average cost as

$$AvgCost(t,s) = \underset{\{a:Solves(a,s)\}}{\text{Avg}} (Cost(t|_a)) \tag{5.7}$$

and assuming a single static ordering, (5.6) becomes

$$\begin{aligned} Cost(t) &= Cost(t_{1,i-1}) + Numsol(t_{1,i-1}) * AvgCost(t_{i,M}, t_{1,i-1}) \\ &= \sum_{i=1}^{M} Numsol(t_{1,i-1}) * AvgCost(p_i, t_{1,i-1}) \,. \end{aligned} \tag{5.8}$$

Assuming that the average cost can be calculated for each individual conjunct, this equation provides a means of calculating the total cost of producing the solutions to a sequence of conjuncts.

Intuitively, (5.5) calculates the number of leaf nodes in a search space and (5.8) sums up the costs associated with producing each node in the search space. Note that none of these equations depend on the absence of functional expressions or on the assumption that no inference is involved. These assumptions come into play in computing the cost for an individual conjunct.

5.2.2 The Cost of Solving a Conjunct

The cost of producing the solutions to a particular conjunct is the sum of the costs of producing each individual solution. Assuming that all of the answers for a conjunct are directly available in the problem solver's database, the cost of producing solutions to a particular conjunct will be the product of the number of solutions to the conjunct and the average cost of producing each solution.

$$Cost(p) = \Delta + K * Numsol(p) \tag{5.9}$$

Here K is the average cost of producing each solution and Δ is the overhead associated with determining whether the proposition p has any solutions. We have assumed that Δ and K are constants and properties of the system's database. In other words, we are assuming that Δ and K are independent of the particular conjunct being solved.

Fortunately, the constants Δ and K have no net effect on the conjunct ordering process because they can be factored out of all the cost equations. To see this, let the *adjusted cost* for a sequence of conjuncts be

$$Cost'(t) = \frac{Cost(t) + \Delta(Numsol(t) - 1)}{\Delta + K} . \tag{5.10}$$

For any two orderings of the same set of conjuncts, the adjusted cost for one will be lower than the adjusted cost for the other, if and only if the same relationship holds for their costs. This can be seen from the above definition, since the number of solutions to a set of conjuncts is independent of their ordering. Therefore, the adjusted cost will serve just as well for determining conjunct order. We need to show, however, that the adjusted cost and average adjusted cost for a single conjunct are independent of Δ and K and that the cost equations (5.8) are independent of Δ and K.

Substituting (5.9) into the above definition, the adjusted cost for a single conjunct is

$$\begin{aligned} Cost'(p) &= \frac{\Delta + K * Numsol(p) + \Delta(Numsol(p) - 1)}{\Delta + K} \\ &= Numsol(p) \end{aligned} \tag{5.11}$$

that is independent of Δ and K. Defining the average adjusted cost as before, the average adjusted cost for a sequence of conjuncts and for a single conjunct become

$$\begin{aligned} AvgCost'(t,s) &= \underset{\{a:Solves(a,s)\}}{\text{Avg}} (Cost'(t|_a)) \\ &= \frac{AvgCost(t,s) + \Delta(AvgNumsol(t,s) - 1)}{\Delta + K} \end{aligned} \tag{5.12}$$

and

$$\begin{aligned} AvgCost'(p,s) &= \underset{\{a:Solves(a,s)\}}{\text{Avg}} (Cost'(p|_a)) \\ &= AvgNumsol(p,s) . \end{aligned}$$

Again, the latter is independent of Δ and K.

To show that the cost equations remain independent of Δ and K, we solve (5.10) and (5.12) for *Cost* and *AvgCost* and substitute into the cost equation (5.8). This

Individual conjunct

$$Cost'(p) = Numsol(p)$$
$$AvgCost'(p,s) = AvgNumsol(p,s)$$

Conjunct sequence

$$\begin{aligned} Cost'(t) &= Cost'(t_{1,i-1}) + Numsol(t_{1,i-1}) * AvgCost'(t_{i,M}, t_{1,i-1}) \\ &= \sum_{i=1}^{M} Numsol(t_{1,i-1}) * AvgCost'(p_i, t_{1,i-1}) \\ &= \sum_{i=1}^{M} Numsol(t_{1,i-1}) * AvgNumsol(p_i, t_{1,i-1}) \\ &= \sum_{i=1}^{M} Numsol(t_{1,i}) \end{aligned}$$

Figure 5.2: Adjusted cost equations

gives

$$Cost(t) = Cost(t_{1,i-1}) + Numsol(t_{1,i-1}) * AvgCost(t_{i,M}, t_{1,i-1})$$

$$\begin{aligned} &(\Delta + K)Cost'(t) + \Delta(Numsol(t) - 1) \\ &\quad = (\Delta + K)Cost'(t_{1,i-1}) + \Delta(Numsol(t_{1,i-1}) - 1) \\ &\qquad + Numsol(t_{1,i-1})((\Delta + K)AvgCost'(t_{i,M}, t_{1,i-1}) \\ &\qquad\qquad + \Delta(AvgNumsol(t_{i,M}, t_{1,i-1}) - 1)) \end{aligned}$$

$$\begin{aligned} &(\Delta + K)Cost'(t) \\ &\quad = (\Delta + K)Cost'(t_{1,i-1}) \\ &\qquad + (\Delta + K)Numsol(t_{1,i-1})AvgCost'(t_{i,M}, t_{1,i-1}) \end{aligned}$$

$$Cost'(t) = Cost'(t_{1,i-1}) + Numsol(t_{1,i-1}) * AvgCost'(t_{i,M}, t_{1,i-1}) .$$

The cost equations (5.8) therefore remain unchanged for adjusted cost. Because of this, we will use adjusted cost for the remainder of this paper. The equations for adjusted cost are summarized in Figure 5.2 for easy reference. With these simplifications, the characteristics of the cost equation can be easily seen. If the search space for the problem continues to expand dramatically with each conjunct the net cost will be proportional to the total number of solutions. If, however, the space first expands and then undergoes dramatic reduction (as in most conjunctive problems), one of the intermediate terms will dominate. This corresponds to the widest portion of the search tree.

5.2.3 The Average Number of Solutions to a Conjunct

Using information about the number of solutions to individual conjuncts and the definition of average number of solutions, we could theoretically compute the average number of solutions for any desired set of conjuncts. However, this would be expensive and pointless, since we would have to find all of the solutions to the various subsequences of conjuncts in order to compute the averages required. If these solutions were known there would be no reason to reorder the conjuncts.

It is sometimes possible to estimate or infer these averages based on available information. The easiest case is when every member of the desired set of conjuncts has the same number of solutions and this number is known. In this case the average number of solutions will be that same value.

$$(q \Longrightarrow Numsol(p) = n) \Longrightarrow AvgNumsol(p, q) = n \tag{5.13}$$

As an example of how this can be used, consider the conjunctive problem

$$Senator(x) \wedge Resident(x, California) \wedge Parent(y, x)$$

where we want to compute the cost of solving the problem. Among other things we need to determine

$$AvgNumsol(\text{“}Parent(y, x)\text{”}, \text{“}Resident(x, California) \wedge Senator(x)\text{”}) .$$

Using the common knowledge that every person has two natural parents

$$Person(\chi) \Longrightarrow Numsol(\text{“}Parent(y, \chi)\text{”}) = 2$$

and that senators are people

$$Senator(x) \Longrightarrow Person(x)$$

we get

$$Senator(\chi) \wedge Resident(\chi, California) \Longrightarrow Numsol(\text{“}Parent(y, \chi)\text{”}) = 2 .$$

Using (5.13) we get the desired conclusion

$$AvgNumsol(\text{“}Parent(y, x)\text{”}, \text{“}Senator(x) \wedge Resident(x, California)\text{”}) = 2 .$$

Thus, using information about the implications of previous conjuncts can permit direct computation of the average number of solutions to a conjunct.

A second case where the average number of solutions to a set of conjuncts can be computed is when the set can be broken up into two smaller sets and the average

number of solutions can be computed for these smaller sets. In general we know that

$$\underset{s_1 \cup s_2}{\text{Avg}}(f) = \frac{\underset{s_1}{\text{Avg}}(f)\|s_1\| + \underset{s_2}{\text{Avg}}(f)\|s_2\| - \underset{s_1 \cap s_2}{\text{Avg}}(f)\|s_1 \cap s_2\|}{\|s_1 \cup s_2\|}$$

where $\|s\|$ refers to the cardinality of the set s. Applying this to the average number of solutions gives

$$AvgNumsol(p, q_1 \vee q_2) = \frac{\begin{array}{l} AvgNumsol(p, q_1)Numsol(q_1) \\ \quad + AvgNumsol(p, q_2)Numsol(q_2) \\ \quad - AvgNumsol(p, q_1 \wedge q_2)Numsol(q_1 \wedge q_2) \end{array}}{Numsol(q_1 \vee q_2)} . \tag{5.14}$$

Thus $AvgNumsol(p, q)$ can be computed in cases where q can be broken into two disjoint pieces q_1 and q_2, and the average number of solutions to p is known for these two sets.

A useful special case of this is when $p \Longrightarrow q$. In this case, if q_1 is taken to be p and q_2 is taken to be disjoint with q_1

$$\begin{array}{rcl} AvgNumsol(p, q_1) & = & AvgNumsol(p, p) = 1 \\ AvgNumsol(p, q_2) & = & AvgNumsol(p, \neg p) = 0 \end{array}$$

that gives

$$(p \Longrightarrow q) \Longrightarrow AvgNumsol(p, q) = \frac{Numsol(p)}{Numsol(q)} . \tag{5.15}$$

As an example, if we know that

$$\begin{array}{l} CurrentDirectory(u) \Longrightarrow Directory(u) \\ Numsol(\text{“}CurrentDirectory(u)\text{”}) = 1 \\ Numsol(\text{“}Directory(u)\text{”}) = 100 \end{array}$$

then (5.15) applies giving

$$AvgNumsol(\text{“}CurrentDirectory(u)\text{”}, \text{“}Directory(u)\text{”}) = \frac{Numsol(\text{“}CurrentDirectory(u)\text{”})}{Numsol(\text{“}Directory(u)\text{”})} = \frac{1}{100} .$$

If none of the above special cases apply (the number of solutions is not the same for every member of p and q cannot be broken into two useful subsets), an assumption must be made in order to compute the average number of solutions. In such cases the most reasonable assumption is that $AvgNumsol(p, q)$ is the same as

the average number for some larger set q' containing q. This assumption can be stated formally as

$$(q \Longrightarrow q') \wedge AvgNumsol(p, q') = n \stackrel{d}{\Longrightarrow} AvgNumsol(p, q) = n \tag{5.16}$$

where the notation $p \stackrel{d}{\Longrightarrow} q$ means that if p is true, and q is consistent, q can be assumed.[3]

Extending the example above, suppose our object were to compute the average number of solutions to the last conjunct in the intelligent agent's problem

$$AvgNumsol(\text{“}CurrentDirectory(u)\text{”}, \text{“}\ldots \wedge DirectoryGroup(u, g)\text{”}) . \tag{5.17}$$

In this case $CurrentDirectory(u)$ does not necessarily imply the other conjuncts. There is also no useful way to divide the solutions to the other conjuncts into two sets, and no exact information is available. However, using the facts that

$$\ldots \wedge DirectoryGroup(u, g) \Longrightarrow Directory(u)$$

and

$$AvgNumsol(\text{“}CurrentDirectory(u)\text{”}, \text{“}Directory(u)\text{”}) = \frac{Numsol(\text{“}CurrentDirectory(u)\text{”})}{Numsol(\text{“}Directory(u)\text{”})} = \frac{1}{100}$$

the above default allows the desired conclusion

$$AvgNumsol(\text{“}CurrentDirectory(u)\text{”}, \text{“}\ldots \wedge DirectoryGroup(u, g)\text{”}) = \tfrac{1}{100} . \tag{5.18}$$

With the knowledge that the average number of solutions can at least be estimated from more primitive number of solutions information, we now turn to the issue of computing that primitive information for individual conjuncts.

5.2.4 Computing the Number of Solutions to a Conjunct

The number of solutions for an individual conjunct can frequently be determined from available information about the sizes of sets. If σ is the set corresponding to the solutions to a proposition ϕ, the number of solutions to the proposition will be the cardinality of the set σ

$$(\beta \in \sigma \Longleftrightarrow \phi|_\beta) \Longrightarrow Numsol(\phi) = \|\sigma\| \tag{5.19}$$

[3]These are sometimes referred to as *normal* defaults. A discussion of such defaults can be found in [Rei80].

Exact Information

It is not uncommon to have exact information about the number of solutions to a problem, even when the bindings for some of the variables are not yet known. Take, for example, the problem of finding the parents of a particular individual. Every individual has exactly two parents and the problem therefore has two solutions. It is not at all unreasonable to expect an intelligent system to have some information of this sort. In the case of the intelligent agent there are several pieces of relevant exact information. For example, $CurrentDirectory$, $DirectoryOf$ and $Size$ are all function symbols. Ground functional expressions (i.e., expressions that do not contain variables) only have a single solution. We can formally express this information to the intelligent agent as

$$\begin{array}{l} Function(Size) \\ Function(DirectoryOf) \\ Function(CurrentDirectory) \end{array} \tag{5.20}$$

and

$$Function(r) \wedge Variable(v) \wedge Ground(\vec{x}) \Longrightarrow \| \{v : r(\vec{x}, v)\} \| = 1 \, . \tag{5.21}$$

An additional piece of information that is useful to the intelligent agent is knowledge that there are an infinite number of solutions to the conjunct $s > 100$. This fact of arithmetic can be stated generally as

$$\| \{x : x > n\} \| = \infty \, . \tag{5.22}$$

Average Information

There are equally many situations where exact information is not available, but average values are available. Take, for example, the problem of determining the children of a given person. The upper bound on the number of children a person could theoretically produce is quite large, especially for males. However, the average number is between two and three. For purposes of estimating the cost of solving a problem, and ordering conjuncts, the average number is much more valuable than an upper bound would be. For the example above, we could express such average information as

$$\begin{array}{c} \underset{\{y:Adult(y)\}}{\text{Avg}} \; (\|Children(y)\|) = 2.3 \\ \underset{\{y:Adult(y) \wedge Catholic(y)\}}{\text{Avg}} \; (\|Children(y)\|) = 3.5 \, . \end{array}$$

Average information can sometimes be computed using information about a relation and its domain. For example, suppose that there are one hundred tuples

that satisfy the relation R, and hence the proposition $R(x,y)$, and that the domain of the first argument, x contains twenty elements. Then, on the average, given x, the number of y's that will satisfy the proposition is five. For binary relations, this relationship can be stated as

$$\| \{x,y : r(x,y)\} \| = n \wedge \|Domain(1,r)\| = d$$
$$\Longrightarrow \underset{x \in Domain(1,r)}{\text{Avg}} (\| \{y : r(x,y)\} \|) = n/d \,.$$

More generally, for relations of arbitrary arity

$$\| \{v : r(\vec{x})\} \| = n \wedge x_i \in v \wedge \|Domain(i,r)\| = d$$
$$\Longrightarrow \underset{x_i \in Domain(i,r)}{\text{Avg}} (\| \{v - \{x_i\} : r(\vec{x})\} \|) = n/d \,. \tag{5.23}$$

As an example, consider the conjunct $Format(f, Scribe)$ from the intelligent agent's problem. Suppose we know that there are 10^4 files on the machine, and 500 of them are Scribe files. The chance that a given file will be a Scribe file is therefore $500/10^4$ or $1/20$.

Approximations and Bounds

When exact information or averages are not available, approximate information or bounds are sometimes valuable. For approximate values, a bound on the error of the approximation is necessary in order for the information to be useful. A very useful special case of approximations is the order of magnitude approximation. When we say that a quantity has order of magnitude n we mean that it is within a factor of ten of the number n. We use the function symbol O for the order of magnitude of a quantity

$$O(v) = n \Longleftrightarrow 0.1n < v \leq 10n \,. \tag{5.24}$$

Much of the remaining interesting information for the intelligent agent's problem is order of magnitude information.

$$\begin{aligned} &O(\|Files\|) = 10^4 \\ &O(\|Directories\|) = 10^2 \\ &O(\| \{g : DirectoryGroup(d,g)\} \|) = 1 \\ &O(\| \{d : DirectoryGroup(d,g)\} \|) = 10 \\ &O(\| \{f : DirectoryOf(f,d)\} \|) = 50 \end{aligned} \tag{5.25}$$

Upper and lower bounds are stated in the normal way.

$$10^2 < \|Directories\| < 10^3$$
$$10^4 < \|Files\|$$

A fact about bounds, which is generally useful, is that ground clauses have at most one solution. If a ground clause is true, there is exactly one solution, otherwise there is no solution.

$$Ground(c) \Longrightarrow \| \{\emptyset : c\} \| \leq 1 \tag{5.26}$$

In the previous section domain information could sometimes be used to derive average values. In the opposite manner, domain information can be used to give upper bounds for an otherwise unknown conjunct. In general, the number of solutions to a conjunct must always be less than the product of the sizes of domains of the unbound variables.

$$\| \{v : r(\vec{x})\} \| \leq \prod_{\{i:Variable(x_i)\}} \|Domain(i,r)\| \tag{5.27}$$

As an example, consider the conjunct $DirectoryGroup(d,g)$ from the intelligent agent's problem. If there are one hundred directories in the machine, and only five groups, the maximum number of solutions to this conjunct would be five hundred.

Using an upper bound on the number of solutions gives an upper bound for the cost of solving a particular conjunct, which leads to an upper bound on the cost of solving an entire problem. This is still useful information if it indicates that one particular ordering of conjuncts is cheaper than any other. Similarly, lower bound information is valuable if it indicates that some conjunct ordering is more expensive than some other. Approximate and order of magnitude values can be used more liberally as estimates, but the final estimate of the cost of solving the problem may be in error by a similar factor.

In order of descending preference, we will use exact information, average values for larger sets, order of magnitude approximations, and lastly, upper and lower bounds for computing the number of solutions and cost information. Using the default notation introduced earlier, these can be stated formally as

$$(q \Longrightarrow q') \wedge AvgNumsol(p,q') = n \stackrel{d}{\Longrightarrow} AvgNumsol(p,q) = n$$
$$O(Numsol(q)) = n \stackrel{d}{\Longrightarrow} Numsol(q) = n$$
$$Numsol(q) \leq n \stackrel{d}{\Longrightarrow} Numsol(q) = n\ .$$

5.2.5 An Example

Using information about set sizes and the cost equations given in Figure 5.2 it is possible to compute the cost of solving a problem for any given ordering of the conjuncts.

As an illustration, Table 5.1 shows the calculations for the conjuncts (in the order given) in the intelligent agent's problem (5.1). Each row of this table gives data for the next conjunct in the sequence, assuming that the conjuncts listed above it

Conjunct	Bound Variables	Number of Solutions	$Numsol(t_{1,i})$	Justification
$File(f)$		$\approx 10^4$	10^4	5.25
$Format(f, Scribe)$	f	≤ 1	10^4	5.26
$Size(f, s)$	f	1	10^4	5.20
$s > 100$	s	≤ 1	10^4	5.26
$DirectoryOf(f, d)$	f	1	10^4	5.20
$DirectoryGroup(d, g)$	d	≈ 1	10^4	5.25
$DirectoryGroup(u, g)$	g	≈ 10	10^5	5.25
$CurrentDirectory(u)$	u	.01	10^3	5.18

Table 5.1: Cost calculations for the intelligent agent's problem

have been processed. The columns contain the list of variables bound by previous conjuncts, the number of solutions to the conjunct as calculated using one of the axioms of the previous sections, and finally the total number of solutions of the conjunct sequence so far (the product of the number of solutions of all conjuncts processed). For example, the conjunct $s > 100$ is the fourth conjunct in the sequence. When it is encountered, the variable s will have been bound by solution of the previous conjunct. The number of solutions will be one (as given by (5.26)), since it is a ground clause. The total number of solutions to the first four conjuncts will be the product of the number of solutions to each of the first four conjuncts, which is 10^4.

From the revised cost equation (Figure 5.2) the sum of the number of solutions to each of subsequences $t_{1,i}$ will be the total cost of processing the conjuncts in the order shown. Using the information in the table above the cost is

$$10^4 + 10^4 + 10^4 + 10^4 + 10^4 + 10^4 + 10^5 + 10^3 = 1.61 * 10^5 .$$

5.2.6 Non-atomic Clauses

In the examples above, each of the individual clauses were atomic propositions. What about the cost of finding the solutions to a disjunction, an implication, or a negated expression? For the most part, these are straightforward.

For a disjunction, the cost of finding the solutions is simply the sum of the costs of finding the solutions to each of the disjuncts. Similarly, the number of solutions is the sum of the number of solutions for each individual disjunct, provided that the solutions to the disjuncts are disjoint. If this is not the case, the sum will provide an upper bound on the number of solutions to the disjunction.

For negated expressions, there are several cases. If the negated expression is a ground clause the average number of answers will be less than or equal to one. More specifically, if the average number of solutions for a ground conjunct p is some number $n \leq 1$, the number for $\neg p$ would be $1 - n$. If, however, the negated expression is not a ground clause, and the number of solutions for the (non-negated) expression is finite, the number of solutions to the negated clause is infinite. As an example, the number of solutions to $President(x)$ is a finite number, while the number of things that are not presidents is infinite. If the negated expression is not a ground clause, but the number of solutions to the (non-negated) expression is either unknown or infinite nothing can be said about the number of solutions. It may be infinite (e.g., $\neg Integer(x)$) or it may be finite (e.g., $\neg NonAlphabetic(x)$). Infinite is a good assumption if double negation is unlikely.

Finally, the number of solutions to more complex expressions such as implications, if-and-only-if statements, and negations of complex expressions can be computed by reducing the expression to conjunctive normal form and using the rules developed for conjunction, disjunction and negation.

5.2.7 The Closed-world Assumption

Thus far we have been assuming that the closed-world assumption holds for our problem solving system

$$True(p) \Longrightarrow InDataBase(p) \ .$$

In other words, we have been assuming that the problem solver is capable of completely enumerating the solutions to any of the conjuncts in a problem. This is not always a valid assumption for AI systems. For example, the intelligent agent might be unable to enumerate the solutions to a conjunct such as $DirectoryGroup(d, g)$ unless the directory variable d is bound. Such limitations must be considered in the choice of conjunct order or the answers that result will be incomplete or erroneous. For example, given that there are only about a dozen directories in any given directory group, an agent operating under the closed-world assumption might decide to first expand conjuncts binding the directory group g and then expand the conjunct $DirectoryGroup(d, g)$. This would lead to the incorrect answer that no directories and hence no files satisfied the query. This would merely be frustrating if the user were looking for a particular file. It could be disastrous if the intelligent agent's response were used as the basis of a backup or deletion operation.

For purposes of conjunct ordering, the closed-world assumption can be handled by adding the condition to the cost equation (Figure 5.2) that the conjunct be enumerable

$$Enumerable(p) \Longrightarrow Cost'(p) = Numsol(p)$$

where $Enumerable(p)$ means that the proposition p obeys the closed-world assumption. If a conjunct is not enumerable its cost is taken to be infinite.

$$\neg Enumerable(p) \Longrightarrow Cost'(p) = \infty$$

In this way, conjuncts that are not enumerable will not be chosen for expansion over others that are enumerable and finite.

Finally we include the default that the closed-world assumption is valid except in those cases where explicit information to the contrary has been provided.

$$\stackrel{d}{\Longrightarrow} Enumerable(p)$$

In the case of the intelligent agent we will assume that it cannot enumerate the set of Scribe files, the set of file-size pairs, or the set of file-directory pairs without first enumerating the set of files.

$$\begin{array}{l} \neg Enumerable(\text{“}Format(f,\sigma)\text{”}) \\ \neg Enumerable(\text{“}Size(f,s)\text{”}) \\ \neg Enumerable(\text{“}DirectoryOf(f,d)\text{”}) \\ \neg Enumerable(\text{“}DirectoryGroup(d,g)\text{”}) \end{array} \tag{5.28}$$

This information will prevent an intelligent agent from making potentially disastrous mistakes.

5.3 Ordering

Given the cost axioms (Figure 5.2) we could (theoretically) enumerate all possible orderings of the conjuncts in a problem, compute the cost for each, and choose the minimal one. For a problem with m conjuncts, this requires enumerating and computing the cost for $m!$ orderings. This is not unreasonable for a problem with only four conjuncts (twenty-four orderings to examine, possibly resulting in a savings of several orders of magnitude in the size of the search tree), but for larger problems the cost is prohibitive. For the intelligent agent's problem (5.1) this exhaustive approach would require examining over 40,000 possible conjunct orderings.

How then, can we prune this search space without sacrificing optimality? In the sections that follow, several ordering techniques are examined with this consideration in mind.

5.3.1 Optimal Ordering

The conjunct ordering problem has been shown to be NP-complete [Tre86]. Thus in the worst case, it is necessary to examine a large number of possible conjunct

orderings for a set of conjuncts. Fortunately, we can usually do very much better than this without sacrificing optimality. Using best-first search [BF81] it is possible to search the space in a way that rapidly leads to the optimal solution for most practical sets of conjuncts, and degrades gracefully to $m!$ for a worst case set of conjuncts. The evaluation function used in this search (to decide which branch of the space to explore next) is the total cost of the partial sequence of conjuncts for that branch (as given by the cost equation, Figure 5.2). To illustrate, we begin with a set of "candidate" conjunct sequences, which initially contains only the null sequence. We then choose the candidate of lowest cost and expand it. Expanding a candidate means constructing all possible candidates formed by appending one of the remaining conjuncts (not yet in the sequence) to the chosen candidate. The number of solutions to the new candidate sequence will be the product of the number of solutions to the parent candidate sequence and the number of solutions to the appended conjunct. Its total cost (as given by the cost equations) is easily calculated as the sum of the cost of the parent candidate and the number of solutions for this new candidate. This process repeats.

At each stage in the algorithm, the total cost of the candidate sequence is used to decide which sequence to expand next. If the cheapest candidate is ever a complete sequence (contains all of the conjuncts), the optimal conjunct ordering has been found and the search can be terminated. Formally this is true because our evaluation function is a lower bound on the actual cost of solving a complete sequence and therefore obeys the optimality requirement of the A* algorithm [Nil80]. A flow chart of this algorithm is given in Figure 5.3.

It is definitely true that a better evaluation function would further improve this algorithm. Ideally, the evaluation function should be the sum of the cost for the candidate and some good estimate of the minimum cost of solving the remaining conjuncts. Unfortunately, no such estimate exists. About the best that we could do is to multiply the number of outstanding conjuncts by some weighting factor (say some fraction of the number of solutions to the problem so far) and add this to the computed cost. This might improve the performance in some cases, but sacrifices the guarantee of optimality. A minor enhancement, that still retains optimality, would be to add the number of outstanding conjuncts to the cost for a candidate. This estimate satisfies the lower bound requirement since the added cost for each of the remaining conjuncts will be at least one.

Several other optimizations can be performed on the algorithm. The first three of these optimizations are syntactic, while the final three depend upon particular properties of the conjunct ordering problem.

- The candidate set can be kept as an ordered list.
- When expanding a candidate, only its cheapest successor need be added to

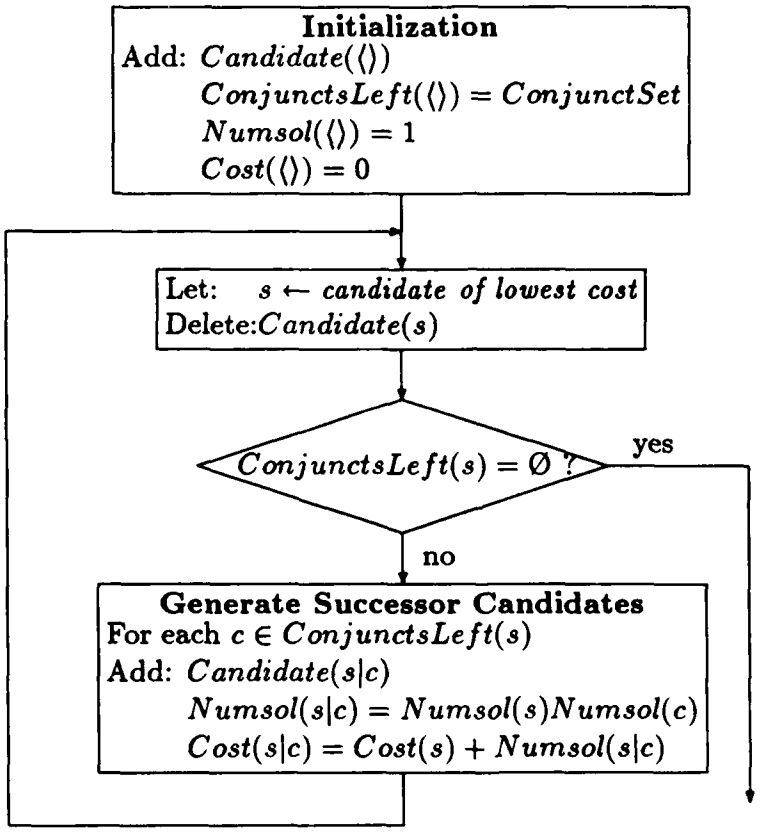


Figure 5.3: Best-first search of the conjunct ordering space

the candidate set immediately. If that successor is ever expanded, the next cheapest successor must be added to the candidate set. This is a form of "partial expansion" of a candidate (see [Nil80], page 67) equivalent to leaving the expanded candidate on the candidate list but updating its "cost" to reflect the next conjunct.

- If a candidate is generated that is simply a permutation of an existing candidate (i.e., the same remaining set of conjuncts), only the cheapest of the two candidates need be preserved. This optimization is a general one for searching graphs rather than trees and is discussed at length in [Nil80], page 64ff. Note that no "cost readjustment" will be necessary in discarding candidates. Either the existing candidate will be cheaper, or the existing candidate won't be expanded yet. This is because the evaluation function is "monotonic" (see [Nil80], page 81ff).

- If a conjunct is not practically enumerable, or has an infinite set size it need not be considered in generating candidates. In other words, candidates whose cost is too large can be thrown away.

- The most expensive conjunct at each stage need not be considered (Theorem 5.2 *in the next section*).

- When expanding a candidate, if the cheapest conjunct actually reduces the search space (its average number of solutions is less than one) only that successor candidate need be added to the list (Theorem 5.1 in the next section).

The final three optimizations can significantly reduce the size of the search space for practical conjunct ordering problems. By looking at the data for the intelligent agent's problem (Table 5.2) we can see that the space searched by this algorithm is not nearly as large as what might be expected. For this problem, only 18 out of the total of 8! possible orderings are ever explored by this algorithm. This sort of behavior is quite common because practical problems often contain many conjuncts that are not immediately enumerable, or that produce a very large number of solutions. In the intelligent agent's problem only two of the eight conjuncts are enumerable to begin with and one of these is quite expensive. Thus, finding an optimal ordering is often not as expensive as might be suspected.

5.3.2 Cheapest-first

One common heuristic for ordering conjuncts is to take the "cheapest" conjunct next at each stage in the ordering process.[4] More precisely, the cheapest conjunct

[4]Kowalski refers to this ordering rule as the "Principle of Procrastination" [Kow79].

Stages of the Conjunct Ordering								
Conjunct	1	2	3	4	5	6	7	8
$CurrentDirectory(u)$	1							
$DirectoryGroup(u,g)$	–	1						
$DirectoryGroup(d,g)$	–	–	10					
$DirectoryOf(f,d)$	–	–	–	50				
$Size(f,s)$	–	–	–	–	1			
$s > 100$	∞	∞	∞	∞	∞	1		
$Format(f, Scribe)$	–	–	–	–	1	1	1	
$File(f)$	10^4	10^4	10^4	10^4	1	1	1	1

Table 5.2: Number of solutions per conjunct at each stage of the ordering process ("-" indicates that the conjunct was not enumerable).

is chosen first, then, assuming that those variables will be bound, the cost of each remaining conjunct is evaluated, and the process is repeated. Using the vocabulary of Section 5.1.1 this ordering criterion can be expressed as

$$\begin{aligned} &CheapestConjunct(s,x) \wedge BestOrdering(s - x|_{Variables(x)}, t) \\ &\quad \Longrightarrow BestOrdering(s, x|t) \end{aligned} \tag{5.29}$$

where the cheapest conjunct is the one of lowest cost according to the equations of the previous section

$$CheapestConjunct(s,x) \Longleftrightarrow x \in s \wedge \forall y \in s\,(Cost'(x) \leq Cost'(y)) \ .$$

If we apply this ordering rule to the intelligent agent's problem, the cheapest initial conjunct is the clause $CurrentDirectory(u)$ that has only one solution (from (5.20) and (5.21)). With the variable u bound, $DirectoryGroup(u,g)$ proves to be the cheapest of the remaining conjuncts having on the order of one solution (using (5.25)). With the variable g bound, the next cheapest is $DirectoryGroup(d,g)$ having on the order of ten solutions (using (5.25)). Table 5.2 summarizes the results of this process. The conjuncts are listed in the order that they would be chosen, using the cheapest-first heuristic, and the columns show the cost of each remaining conjunct at each step in the ordering process. From the table, we can see that the cheapest-first heuristic leads to an optimal ordering of the conjuncts for this problem.

While this heuristic works very well for this particular problem, there are many simple examples where it fails miserably. The difficulty is that this heuristic will not necessarily focus the problem solving effort. Instead, it may direct the problem solver to jump helter-skelter about the problem, picking off the easy tidbits. All

the while, the size of the search space increases and the inevitable tough part of the problem is simply delayed. Sometimes this sort of opportunism will help solve the tough part of a problem, but often it does not. In fact, solving the tough part of the problem first may very well make everything else trivial to solve. This is where the cheapest-first heuristic can lead to totally unacceptable orderings.

As an example, consider the problem of finding all of the files on a particular directory that occupy less than ten pages

$$DirectoryOf(f, Genesereth) \wedge Size(f, s) \wedge s < 10$$

According to the cheapest-first heuristic the conjunct $s < 10$ is cheapest, and is selected first. Following this, the conjunct $DirectoryOf(f, Genesereth)$ would be selected leaving the conjunct $Size(f, s)$. According to the cost equation, the cost of solving the conjuncts in this order is $\approx 10^3$. Solving them in the order given, however, results in a cost $\approx 10^2$. In this case, the cheapest-first heuristic increases the cost by a factor of ten. The penalty could easily be much higher.

The cheapest-first heuristic fails because the final conjunct does not help in the solution of either the first or second conjuncts, which constitute the "crux" of the problem. Once the first two conjuncts are solved, the final conjunct becomes a ground clause and costs little to solve. As a result there is no advantage to solving the final conjunct first, even though it is the cheapest.

While general use of the cheapest-first heuristic may cause serious difficulties, there are two special cases where it is valuable.

Theorem 5.1 *When the cheapest conjunct will reduce the search space rather than expand it (i.e. the average number of solutions is less than one), this conjunct should be chosen first. The resulting ordering will always have a cost less than twice that of the optimal ordering and will be the optimal ordering if the problem is non-trivial.*

Proof *Consider a set of conjuncts whose cheapest conjunct c has an expected number of solutions less than one ($Numsol(c) < 1$). Suppose that some ordering $t = t_{1,i-1}|c|t_{i+1,M}$ of the conjuncts is optimal. Using the cost equations (Figure 5.2) the cost of solving t is*

$$\begin{aligned} Cost'(t) = Cost'(t_{1,i-1}) &+ Numsol(t_{1,i-1}) AvgCost'(c, t_{1,i-1}) \\ &+ Numsol(t_{1,i-1}|c) AvgCost'(t_{i+1,M}, t_{1,i-1}|c) \,. \end{aligned} \tag{5.30}$$

Alternatively, if the conjunct c were placed first in the sequence giving $t' = c|t_{1,i-1}|t_{i+1,M}$ the cost would be

$$\begin{aligned} Cost'(t') = Cost'(c) &+ Numsol(c) AvgCost'(t_{1,i-1}, c) \\ &+ Numsol(c|t_{1,i-1}) AvgCost'(t_{i+1,M}, c|t_{1,i-1}) \,. \end{aligned} \tag{5.31}$$

The final terms in these two equations are identical, since the number of solutions to $c|t_{1,i-1}$ is the same as for $t_{1,i-1}|c$. The other two terms in (5.31) are both smaller than the initial term in (5.30) since

$$Cost'(c) < Cost'(t_{1,i-1})$$
$$AvgCost'(t_{1,i-1}, c) \leq Cost'(t_{1,i-1})$$
$$Numsol(c) < 1 .$$

If $Cost'(t_{1,i-1})$ is small (i.e. less than one), the sequence $t_{1,i-1}|c$ is easy to solve and order makes little difference. In this case the cost of doing c first is no worse than twice the cost for doing $t_{1,i-1}$ first since the first two terms of (5.31) are both smaller than the first term of (5.30). If the second or third terms of (5.30) are significant, the latter ordering will be as good or better.

Alternatively, if $Cost'(t_{1,i-1})$ is not small, the first term of (5.31) becomes insignificant since

$$Cost'(c) \ll Cost'(t_{1,i-1}) .$$

In addition the second term of (5.31) will be smaller than the first term of (5.30)

$$Numsol(c)AvgCost'(t_{1,i-1}, c) < Cost'(t_{1,i-1}) .$$

The cost of solving t' will therefore be strictly less than the cost of solving the original ordering t. As a result, the optimal ordering must begin with the conjunct c in this case. □

Formally, the ordering principle of Theorem 5.1 can be expressed as

$$\begin{aligned} &CheapestConjunct(s, x) \wedge Cost'(x) \leq 1 \\ &\qquad \wedge BestOrdering(s - x|_{Variables(x)}, t) \\ &\Longrightarrow BestOrdering(s, x|t) . \end{aligned} \tag{5.32}$$

This principle is useful for reducing the best-first search described in the previous section.

Another useful special case of the cheapest-first heuristic is the *Adjacency Restriction*.

Theorem 5.2 *(Adjacency Restriction) Suppose that the conjunct sequence $t = t_{1,i-1}|c|d|t_{i+2,M}$ is an optimal ordering of the conjuncts involved. For any adjacent pair of conjuncts c and d the number of solutions to the second conjunct d (under the bindings to the preceding conjuncts $t_{1,i-1}$) must be greater than the number of solutions to the conjunct c (under the bindings to the preceding sequence $t_{1,i-1}$). Formally, $AvgNumsol(c, t_{1,i-1}) \leq AvgNumsol(d, t_{1,i-1})$.*

Proof *According to the cost equations (Figure 5.2)*

$$\begin{aligned} Cost'(t) = Cost'(t_{1,i-1}) &+ Numsol(t_{1,i-1})AvgNumsol(c, t_{1,i-1}) \\ &+ Numsol(t_{1,i-1}|c)AvgNumsol(d, t_{1,i-1}|c) \\ &+ Numsol(t_{1,i-1}|c|d)AvgCost'(t_{i+2,M}, t_{1,i-1}|c|d) \\ = Cost'(t_{1,i-1}) &+ Numsol(t_{1,i-1})AvgNumsol(c, t_{1,i-1}) \\ &+ Numsol(t_{1,i-1}|c|d) \\ &+ Numsol(t_{1,i-1}|c|d)AvgCost'(t_{i+2,M}, t_{1,i-1}|c|d) \ . \end{aligned}$$

Similarly, if $t' = t_{1,i-1}|d|c|t_{i+2,M}$ *(c and d reversed) the cost is*

$$\begin{aligned} Cost'(t') = Cost'(t_{1,i-1}) &+ Numsol(t_{1,i-1})AvgNumsol(d, t_{1,i-1}) \\ &+ Numsol(t_{1,i-1}|c|d) \\ &+ Numsol(t_{1,i-1}|c|d)AvgCost'(t_{i+2,M}, t_{1,i-1}|c|d) \ . \end{aligned}$$

The first, third and fourth terms in each of these equations are identical. Therefore

$$Cost'(t) \leq Cost'(t') \Longleftrightarrow AvgNumsol(c, t_{1,i-1}) \leq AvgNumsol(d, t_{1,i-1}) \ .$$

By our assumption that t is an optimal ordering, the left hand side must be true. Therefore

$$AvgNumsol(c, t_{1,i-1}) \leq AvgNumsol(d, t_{1,i-1}) \ .$$

□

Note that the adjacency restriction is much weaker than the cheapest-first heuristic. It does not imply that a conjunct should be less expensive than all subsequent conjuncts, only that it should be less expensive than its immediate successor. In other words, the adjacency restriction does not necessarily imply that the conjunct c will be cheaper than other conjuncts in $t_{i+2,M}$. Although this restriction may appear weak, it has several useful corollaries.

Corollary 5.3 *The most expensive conjunct is never the optimal one to do next.*

If the most expensive one was selected, any conjunct chosen to follow it would be less expensive, violating the adjacency restriction.

Corollary 5.4 *The search for conjuncts to immediately follow a conjunct c can be limited to those that were more expensive than c at the time c was selected.*

These corollaries significantly reduce the size of the space that must be searched in order to find the optimal ordering. In fact, some simple analysis shows that the number of possible orderings that must be considered by a procedure employing the

n	$F(n)$	$n!$
1	1	1
2	1	2
3	2	6
4	5	24
5	16	120
6	61	720
7	272	5040
8	1385	40,320
9	7936	362,880
10	50,521	3,628,800

Table 5.3: Reduction of the ordering space by the adjacency restriction

adjacency restriction will be bounded by $F(n)$ where n is the number of remaining conjuncts and

$$F(n) = G(n,0)$$

$$G(n,d) = \begin{cases} 0 & \text{if } n = d; \\ 1 & \text{if } n = 1, d = 0; \\ \sum_{i=0}^{n-d-1} G(n-1,i) & \text{otherwise.} \end{cases}$$

Here d can be thought of as the number of remaining conjuncts that cannot appear as the next conjunct because of the adjacency restriction. Note that if the second argument to G is ignored, this formula reduces to $n!$ as expected. The case of $i = 0$ corresponds to the case where the cheapest remaining conjunct is chosen. In this case any other remaining conjunct will be allowed in the next position. Similarly, the case of $i = n - 1$ corresponds to choosing the most expensive conjunct. In this case none of the remaining conjuncts will be allowed in the next position (by Corollary 5.4).

In Table 5.3 the values of $F(n)$ and $n!$ are shown for different values of n. For four conjuncts the adjacency restriction reduces the search to only five possible orderings. For eight conjuncts, the space is reduced from over forty thousand possible orderings to only 1385 possible orderings. The figures show that the adjacency restriction significantly reduces the possible worst case search for a best-first search of the space of conjunct orderings.

We will make further use of the adjacency restriction in Section 5.3.4.

5.3.3 Connectivity

Motivated by the faults of the cheapest-first heuristic, one might be tempted to use a connectivity heuristic in deciding which conjunct to do next. By this we mean that each conjunct shares variables with the conjuncts immediately preceding and succeeding it in the sequence. Formally, given a set of conjuncts s we can state this ordering principle as

$$Ordering(s,t) \wedge Connected(t) \Longrightarrow BestOrdering(s,t) \tag{5.33}$$

where

$$\begin{aligned} Connected(t) &\iff (t = t_{1,i-1}|c|d|t_{i+1,M} \Longrightarrow \neg Independent(c,d)) \\ Independent(x,y) &\iff Variables(x) \cap Variables(y) = \emptyset \end{aligned}$$

where $Independent(x,y)$ means that x and y do not share any variables, and $Connected(t)$ signifies that the sequence t is completely connected.

This principle fails for a different reason than the cheapest-first heuristic. Sometimes the best way to find a solution to a difficult conjunct is to attack it from many different sides. For example, consider the problem of finding a printer in a particular building that is capable of printing some special character. Formally this problem can be characterized as finding p and f such that

$$Printerlocation(p, JacksHall) \wedge Font(p,f) \wedge Symbol(f, \bowtie)$$

where $Font(p,f)$ indicates that the printer p has the font f, and $Symbol(f,x)$ indicates that the font f contains the symbol x. Suppose that there are three printers in Jacks' Hall and that each printer has more than fifty fonts, but that only four fonts contain a "bowtie" symbol. According to the connectivity heuristic the ordering given would be acceptable since each conjunct shares variables with its neighbors. According to the cost equations (Figure 5.2) the cost for enumerating this ordering would be greater than

$$3 + 3(50) + 1 = 154\,.$$

Alternatively, if the conjuncts were processed in the optimal order

$$Printerlocation(p, JacksHall) \wedge Symbol(f, \bowtie) \wedge Font(p,f)$$

the cost is bounded by

$$3 + 3(4) + 3(4) = 26\,.$$

Because the conjunct $Font(p,f)$ has many solutions it is cheaper to attack it from two different angles. In general, the connectivity heuristic can also lead to arbitrarily poor orderings.

Since the cheapest-first and connectivity heuristics behave somewhat differently, it is tempting to ask whether these two heuristics taken together can guarantee optimal ordering. Unfortunately, the answer is no. First of all, the connectivity heuristic provides no guidance on which conjunct to start with. As we illustrated in the previous section, the cheapest-first heuristic can suggest a suboptimal starting conjunct. Even assuming that the ordering were started correctly both the connectivity heuristic and the cheapest-first heuristic can suggest and agree upon a conjunct that is irrelevant to solving the crux of the problem. In the previous example suppose that the user were also interested in knowing the queue lengths for the printers involved. The problem then becomes find p, f and l such that

$$Printerlocation(p, JacksHall) \wedge Font(p, f) \wedge Symbol(f, \bowtie) \wedge QueueLength(p, l) .$$

In this case, after the printer location conjunct was enumerated, both the connectivity heuristic and the cheapest-first heuristic would agree on the queue length conjunct (since it is a functional expression). This is definitely suboptimal.

5.3.4 Problem Independence

In the solution of a conjunctive problem, it is not uncommon to find two or more pieces of the problem that are independent. We say that two pieces of a problem are independent if, and only if, they do not share any variables. As an example of independence, consider the intelligent agent's problem (5.1), where the conjuncts are given in the optimal order (as calculated in Section 5.3.2 and shown in Table 5.2)

$$\begin{aligned} CurrentDirectory(u) \wedge\ & DirectoryGroup(u, g) \\ & \wedge DirectoryGroup(d, g) \wedge DirectoryOf(f, d) \\ & \wedge Size(f, s) \wedge s > 100 \wedge Format(f, Scribe) \wedge File(f) . \end{aligned}$$

As in most realistic problems, independence does not arise at the top level of the problem, but only becomes manifest after several of the conjuncts have been solved. In this case, when the fourth conjunct is solved, the variable f is bound and the remaining four conjuncts divide into three independent subproblems

$$\begin{aligned} & Size(f, s) \wedge s > 100 \\ & Format(f, Scribe) \\ & File(f) . \end{aligned}$$

With independent sets of conjuncts the search for an optimal ordering becomes much easier. We conjecture that it is possible to construct the optimal ordering for

the entire collection by determining the optimal orderings for each of the independent sets.[5]

Conjecture 5.5 *If a set of conjuncts divides into two independent sets, the optimal ordering for the entire set will be some interleaving of the optimal orderings for the two independent sets.*

Formally,

$$s = s' \cup s'' \wedge Independent(s', s'') \wedge BestOrdering(s', t') \wedge BestOrdering(s'', t'') \\ \Longrightarrow \exists t\,(Interleaving(t, \{t', t''\}) \wedge BestOrdering(s, t)) \tag{5.34}$$

where

$$Independent(x, y) \Longleftrightarrow Variables(x) \cap Variables(y) = \emptyset\ .$$

To see why interleaving may be necessary consider the problem of finding the solutions to the two independent sets of conjuncts

$$P_1(x) \wedge P_2(x, y)$$
$$P_3(u) \wedge P_4(u, v)\ .$$

Let N_1 and N_3 be the number of solutions to P_1 and P_3 respectively, and let N_2 and N_4 be the average number of solutions to P_2 and P_4 under the bindings of x and u respectively.

$$N_1 = Numsol(P_1(x)) \qquad N_2 = AvgNumsol(P_2(x, y), P_1(x))$$
$$N_3 = Numsol(P_3(u)) \qquad N_4 = AvgNumsol(P_4(u, v), P_3(u))$$

Assume that $P_1(x)$ and $P_3(u)$ both have fewer expected solutions than the other two conjuncts

$$N_1 \leq N_3 \leq N_2 \leq N_4\ .$$

Using the cost equations (Figure 5.2), for the ordering P_1, P_2, P_3, P_4 the cost is

$$N_1 + N_1N_2 + N_1N_2N_3 + N_1N_2N_3N_4\ .$$

However, if the two independent sets are interleaved, P_1, P_3, P_2, P_4, the cost is

$$N_1 + N_1N_3 + N_1N_3N_2 + N_1N_3N_2N_4\ .$$

These costs are identical except for their second terms. Since $N_3 \leq N_2$ the second ordering is superior to the first. It is therefore less expensive to interleave the

[5]While we have been unable to prove this conjecture we have also been unable to construct a counterexample.

solutions of the two independent conjunct sequences than it is to first solve one, then the other. The optimal ordering for this case is therefore

$$P_1(x) \wedge P_3(u) \wedge P_2(x,y) \wedge P_4(u,v) .$$

In general, there is no simple rule that indicates what the interleaving of two independent sequences should be. However, the adjacency restriction (Theorem 5.2) drastically limits the number of possible interleavings.

Corollary 5.6 *If* $t' = t'_{1,i-1}|t'_i|t'_{i+1,M}$ *and* $t'' = t''_{1,j-1}|t''_j|t''_{j+1,N}$ *are two optimally ordered independent sequences of conjuncts containing conjuncts* t'_i *and* t''_j *respectively, and the optimal ordering for the union of the two sets is* $t = t_{1,i+j-2}|t'_i|t''_j|t_{i+j+2,M+N}$, *then*

$$AvgNumsol(t'_i, t'_{1,i-1}) \leq AvgNumsol(t''_j, t''_{1,j-1}) .$$

This follows directly from the adjacency restriction but is stronger because the conjuncts in the two sequences are independent. A consequence of this result is that if the beginning conjunct of one sequence has fewer solutions than all conjuncts in the other sequence (i.e., $\forall j Numsol(t'_1) \leq AvgNumsol(t''_j, t''_{1,j-1})$), it must be the first conjunct in the interleaving. Likewise, if the final conjunct in one sequence has more solutions than all conjuncts in the other sequence (i.e., $\forall j Numsol(t'_M) \geq AvgNumsol(t''_j, t''_{1,j-1})$), it must be last in the interleaving. It is also relatively simple to show that any sequence of conjuncts having a monotonically decreasing sequence of average number of solutions (i.e., $AvgNumsol(t'_i, t'_{1,i-1}) \geq AvgNumsol(t'_{i+1}, t'_{1,i})$) will not be split by interleaving.

In the example above, these consequences of the adjacency restriction limit the field to only one possible interleaving. P_1 must come first because N_1 is smaller than both N_3 and N_4. Similarly, P_4 must come last because N_4 is larger than both N_1 and N_2. Finally, P_3 must come before P_2 because the two conjuncts are adjacent and $N_3 \leq N_2$.

Although interleaving is necessary in theory, we have been unable to find any practical examples where it is necessary or even advantageous. There is a good reason for this. *Independent problems, each of which produce multiple answers, are almost always posed as separate queries.* As a result, the optimal ordering for independent sets of conjuncts is almost always a simple concatenation of the optimal orderings for each of the independent pieces. This heuristic can be stated formally as

$$s = s' \cup s'' \wedge Independent(s', s'') \wedge BestOrdering(s', t') \wedge BestOrdering(s'', t'') \\ \Longrightarrow BestOrdering(s, t'|t'') \vee BestOrdering(s, t''|t') .$$

This rule can significantly reduce the cost of finding solutions to independent sets of conjuncts. Given a set of conjuncts that divides into n independent pieces,

each consisting of k_i conjuncts, the ordering cost is at worst

$$n! + \sum_{i=1}^{n} k_i!$$

as opposed to

$$\left(\sum_{i=1}^{n} k_i\right)!$$

if all possible orderings are considered. For a problem containing ten conjuncts that divide into two independent sets of five conjuncts each, this amounts to a potential savings of over four orders of magnitude in the cost of finding the optimal ordering.

5.4 Extensions

Throughout this paper we have made several limiting assumptions about the nature of the generate and test engine, the characteristics of the database, and the characteristics of conjunctions given to the problem solver. In this section we consider what is necessary to relax these assumptions and discuss some of the open research questions involved in these extensions.

5.4.1 Augmenting the Problem Solver

Throughout this paper, we have been assuming a very simple generate and test mechanism. Several more sophisticated mechanisms have been proposed and used in various AI systems, often with great success. Though it is not our objective to present or analyze such problem solving methods, the use of any of these methods will have a subtle effect upon the enterprise of conjunct ordering. The cost equations developed in Section 5.2 are only valid for the simple generate and test mechanism. Different generate and test mechanisms have different cost equations and different cost equations give rise to different optimal orderings.

Most augmented generate and test procedures take advantage of subproblem independence to eliminate unnecessary backtracking. As an example of such a method, consider a generate and test engine embellished so that whenever a sequence of conjuncts divides into two independent subsequences, the problem solver finds the solutions to each piece independently (as if on separate processors) and then produces the cross product of the two solution sets (i.e. every solution to one subproblem is paired with every solution to the other subproblem).

As an example, consider the problem of finding all of the students who may have

had access to a set of files. The conjunct sequence is

$$\begin{aligned} &DirectoryOf(f, Genesereth) \wedge NameField(f, CS223exam) \\ &\quad \wedge Student(s, CS223) \wedge HasAccountOn(s, SUMEX) \\ &\quad \wedge Username(s, u) \wedge Access(u, f) \ . \end{aligned}$$

The first two conjuncts of this problem are independent of the next three conjuncts in the problem, so our augmented problem solver would independently find the indicated exam files and the students in the class that have accounts on the requisite machine. It would then take the cross product of the two solution sets and try these solutions in the final conjunct to determine which student had access to which files.

The cost of this operation is the sum of the costs of finding the solutions to each independent subsequence plus the cost of constructing the cross product of the solution sets. The cost equation for this problem solver would therefore be

$$\begin{aligned} &t = t_{1,i-1} | t_{i,M} \wedge Independent(t_{1,i-1}, t_{i,M}) \\ &\quad \Longrightarrow Cost'(t) = Cost'(t_{1,i-1}) + Cost'(t_{i,M}) + Numsol(t_{1,i-1}) Numsol(t_{i,M}) \end{aligned}$$

as opposed to

$$Cost'(t) = Cost'(t_{1,i-1}) + Numsol(t_{1,i-1}) Cost'(t_{i,M})$$

for straightforward generate and test.

The issue here is not one of determining which method is superior[6] but rather of finding a cost equation that accurately characterizes the behavior of the problem solver being used. While this was simple enough for the example method given above, it is not necessarily easy in general. One well known generate and test procedure that takes advantage of total independence is *selective backtracking* [PP82].[7]. This method functions like a simple generate and test procedure as long as the search continues successfully. Under these conditions the cost would be described by the cost equations of Section 5.2.1. But when failure occurs, and backtracking takes place, previous conjuncts that are independent of the failing conjunct are

[6] In this case the more sophisticated generate and test mechanism only pays off when $Cost'(t_{i,M}) \gg Numsol(t_{i,M})$ and $Numsol(t_{1,i-1}) \gg 1$. When $Numsol(t_{1,i-1})$ is less than or equal to one, there is no advantage to breaking the problem into independent pieces since solving $t_{1,i-1}$ first will actually eliminate solutions that would otherwise have to be tried in $t_{i,M}$.

[7] There is a strong connection between selective backtracking and dependency-directed backtracking [SS77]. It is possible to view selective backtracking as a special case of dependency-directed backtracking or reason maintenance [Doy79] applied to explicit meta-level statements about the satisfiability of the current goals of a problem solver. New goals for a problem solver depend upon previous variable bindings. Hence, when the assumption that a goal is satisfiable proves false one of the premises that led to that assumption must be false, namely one of its variable bindings must be incorrect.

bypassed by the intelligent backtracking. Thus, the cost equation for failing cases looks more like that in the example above. The actual cost equation for this method is a weighted average of the two different cost equations, where the weighting is a function of the probability of failure for the particular problem at hand.

It would be a mistake to think that ordering and selective backtracking (or any other more sophisticated generate and test mechanism) are alternative techniques. They are not. In fact, they are largely orthogonal. The intelligent agent's problem (5.1) serves to illustrate that selective backtracking is no substitute for conjunct ordering. For this sequence of conjuncts, smart backtracking provides virtually no improvement over an ordinary generate and test engine. Yet, as we have demonstrated, conjunct ordering can provide an improvement of two orders of magnitude.

Likewise, conjunct ordering does not eliminate the advantage of selective backtracking. It is true that an optimally ordered set of conjuncts will naturally involve less backtracking than an arbitrarily ordered set since an optimally ordered set of conjuncts describes a smaller search space and therefore contains fewer branches that fail. However, backtracking does occur any time failure occurs, and intelligent backtracking will be of value in those cases where the optimal ordering consists of whittling away at some portion of the problem by attacking it from several different angles. As an example, consider the problem of finding a president having a brother and a sister who live in Massachusetts. The optimally ordered set of conjuncts is

$$\begin{aligned} &President(x) \wedge Brother(x,b) \wedge Resident(b,Mass) \\ &\quad \wedge Sister(x,s) \wedge Resident(s,Mass)\ . \end{aligned}$$

Suppose that we have found a president with a brother that lives in Massachusetts. If this president has no sisters it is pointless to look for additional resident brothers. Selective backtracking is valuable here, since it bypasses the independent brother clause and immediately backtracks to the first clause.

5.4.2 Dynamic Ordering

So far we have been assuming that a single static ordering for a set of conjuncts is optimal for producing all of the solutions. In fact this is not always the case. For example, consider the conjunct

$$DirectoryOf(f,d)$$

where the variable d will be bound at the time the conjunct is to be solved. The average number of solutions to this conjunct is

$$O(\| \{f : DirectoryOf(f,d)\} \|) = 20\ .$$

Yet some directories have very few files while others have hundreds of files. As a result, the optimal ordering of the remaining conjuncts might be different for different bindings of the directory variable. In general, the optimal ordering for a set of conjuncts is an ordering tree, where each branch of the tree represents a different ordering of the remaining conjuncts corresponding to a different set of solutions to the preceding conjuncts in the tree. The branching factor at each level of the tree corresponds to the sensitivity of the solution set size for a particular conjunct to the actual variable bindings resulting from previous conjuncts.

Unfortunately, the techniques developed in Section 5.3 are of little use in determining the optimal ordering tree. They assume a single static ordering, and base that ordering on averages over the set of all solutions to previous conjuncts. If different orderings are used for different solutions to preceding conjuncts, the cost equations of Section 5.2 are not accurate and may not correctly determine even the optimal first conjunct. In general, finding the optimal ordering tree for a set of conjuncts requires searching through the space of all possible ordering trees. This is an enormous space. Furthermore, generating each tree requires knowledge of the actual solutions to each conjunct. Determining the optimal tree therefore requires solving the problem many times over.

The best that can be done is to choose the first conjunct based on the cost equations of Section 5.2, generate the solutions to that conjunct, and then for each solution to that conjunct choose the best next conjunct, and so forth. In other words, at each step in the generate and test process the remaining conjuncts are ordered for each solution to the preceding conjuncts. We refer to this as *dynamic ordering*.

Assuming that the set sizes used in ordering calculations are accurate, dynamic ordering is guaranteed to result in a problem search space that is at least as small as that produced by the initial (static) ordering. In cases where set size is sensitive to the actual variable bindings, the improvement resulting from dynamic ordering can be dramatic.

Unfortunately, dynamic ordering is very costly because it multiplies the complexity of the search space by the complexity of the ordering problem. A solution to this difficulty is to limit dynamic reordering to those cases where it is likely to have a pronounced effect. This requires preserving the number of solutions information (calculated during the initial ordering of the conjuncts) and at each step in the generate and test process, verifying that the actual number of solutions to each of the remaining conjuncts does not vary significantly from the average calculated initially. If the actual number of solutions for some conjunct grossly exceeds the estimated value, dynamic reordering is warranted. Similarly, if some remaining conjunct has far fewer solutions than anticipated, there might be considerable advantage to reordering the remaining conjuncts.

Run-time cost monitoring should also be used in conjunction with any dynamic ordering strategy. For easy or even moderate problems, the expense of reordering is not warranted. But for very hard problems dynamic reordering can make the difference between solution and failure.

5.4.3 Infinite Sets

Finally, suppose that every conjunct has an infinite number of solutions, but only a finite number of answers are needed. Such cases are particularly prevalent in planning problems. For example, a conjunct such as $On(B, x)$ effectively has an infinite number of solutions because the robot arm can move the block B to an infinite number of different locations. In such cases the cost analysis of Section 5.2 does not apply. The assumption that "the cost of finding a single solution will be that fraction of the cost of finding all solutions" is not valid since all of the quantities involved are infinite. In computing cost for this case what must be considered is not the total number of solutions for each of the conjuncts, but rather the likelihood that a solution to one conjunct will satisfy the succeeding conjuncts. In other words, the cost for finding one solution to the conjunction is computed directly by estimating the number of solutions that must be generated to the first conjunct in order to find one that will satisfy succeeding conjuncts. This can be made precise by dividing each term in the cost equations (Figure (5.2)) by the total number of solutions to the conjunction. Simplification then gives a usable cost equation for determining conjunct order in those cases.

5.5 Discussion

5.5.1 Related Work

Cost Calculation In Section 5.2 we developed formulae for calculating the cost of solving a conjunctive query given information about set sizes, and information about the domains of the relations involved. These formulae are similar in spirit to those developed by Warren for the CHAT-80 system [War81]. In fact, the analyses in Sections 5.2.3 and 5.2.4 were inspired by Warren's use of information about the sizes of the domains for the relations involved. Recently, Rowe [Row83] has done extensive analysis on the use of heuristics for inferring new database statistics from an existing body of database statistics. In particular, Rowe makes use of higher order statistical information, like variance, to help in inferring properties about subsets of known sets. This approach warrants further study.

There has been other work within the database community on evaluating the difficulty of conjunctive queries. Summaries of this work can be found in [Ull82]

and [Kin81]. Database work tends to differ from the work here in the criteria used for calculating the cost of solving a conjunct. For example, in the database world, some of the assumptions are

- Databases are extremely large and must be stored on disk rather than kept in memory. As a consequence, the number of disk accesses is the important optimization criterion, not the size of the search space.

- Often databases employ incomplete indexing mechanisms (e.g. indexing only on the relations of facts). For example, the cost of finding the father of a given person can therefore be just as high as the cost of producing all father-progeny pairs in the database. In contrast, we have assumed that the database is completely indexed, which implies that the cost of finding the father of a given person is a constant, and the cost of finding all father-progeny pairs is proportional to the number of such pairs.

- The database can be regarded as a "closed-world" [Rei78a]. The ability to enumerate the solutions to a particular conjunct is never in question.

- Concurrent accesses and updates to the database must be considered. As a result, the amount of time that a given portion of the database is "locked" is an important (sometimes the most important) cost criterion.

Conjunct Ordering Many papers have been published on the subject of optimizing relational database queries. (A good summary of this work can be found in Ullman [Ull82].) While the basic problem is the same in both cases (finding the optimal way of generating the answers satisfying a conjunctive expression) the methodology and assumptions are usually quite different.

In the database community, the language of Relational Algebra [Ull82] has been widely adopted as a means of expressing queries. The algebra is a language for expressing sequences of referencing and combining operations on databases indexed by relation. In general, any query can be implemented in terms of these operations.

A large percentage of query optimization research has been concerned with syntactic transformations of these relational algebra expressions. These optimizations can, and usually do improve the efficiency of such expressions. However these inefficiencies are, to a large extent, introduced by the language of relational algebra. A simple generate and test engine, operating on predicate calculus expressions, automatically includes these optimizations as a side effect of substituting variable bindings into the remaining conjuncts (at each step in the generation process). We take these optimizations entirely for granted. As a result, work on relational algebra has little relevance to the work presented in this paper.

Leaving the work on relational algebra aside, and factoring out differences in the calculation of cost, there are several database systems that employ some simple conjunct ordering heuristics similar to those presented in Section 5.3.

In the INGRES system [Ull82,WY76] a graph decomposition algorithm is used for ordering conjuncts. This algorithm implicitly encodes several ordering heuristics:

- Prefer conjuncts that contain one or more constant arguments.
- Prefer "simply connected" conjuncts (a simply connected conjunct is one whose variables do not appear in any inequality or disjunction).
- Prefer conjuncts having a "small" number of solutions.
- Prefer conjuncts that break the problem into two or more totally independent pieces.

A second system, called SYSTEM R [Ull82,SAC*79], uses a similar set of ordering heuristics:

- Prefer conjuncts containing one or more constant arguments,
- Prefer conjuncts containing one or more arguments that participate in an inequality with a constant,
- Prefer the conjunct having the fewest solutions.

For both of these systems, the combination of heuristics has an effect similar to that of the cheapest-first and connectivity heuristics examined in Section 5.3.

The cheapest-first heuristic was introduced to artificial intelligence by Kowalski [Kow79] (page 93) and later by Moore [Moo75] (page 78ff). Warren describes its use in a PROLOG based natural language system called CHAT-80 [War81].

There are several important differences between the work described above and the work described in this paper. As we pointed out in Section 5.3 there are many cases where simple ordering heuristics fail badly. As a result, we have stressed the importance of searching the space of possible conjunct orderings to determine an optimal ordering. We have also argued that this need not be outrageously expensive if best-first search is employed.

A second difference between the approach taken here and that employed in the work mentioned above is that we have used an explicit declarative language for expressing conjunct ordering strategies, rather than encoding this information procedurally. However, as we mentioned in Section 2.1 this distinction is superficial. It is the techniques and results that are important and not the language of description. Any of the techniques developed in this paper could be encoded procedurally for use in an intelligent system.

However there is one interesting manifestation of our choice of language; predicate calculus has allowed us to express information about set sizes that could not be expressed in the systems described above. Specifically, information about the cost of solving whole groups of problems can be stated in this language. For example, to determine the number of solutions to the problem $Parent(x, y)$ where the variable x is given, a database system would determine how many parent-progeny tuples were in the database, then determine how many different x's there were in these tuples, and divide the two numbers to produce an average value. In contrast, using the approach presented here, we could directly state the general fact that a person has only two parents.

$$Person(\chi) \Longrightarrow Numsol(\text{“}Parent(y, \chi)\text{”}) = 2$$

A final distinction between our work and that of the work described above is our (admittedly preliminary) consideration of issues such as embellished generate and test engines, dynamic ordering, and treatment of infinite sets. These issues have received relatively little attention within the database community.

5.5.2 Final Remarks

We have argued that conjunct ordering is unavoidable for many intelligent systems. For the intelligent agent many realistic requests cannot be solved without recourse to conjunct ordering.

While simple ordering heuristics may work well for the majority of cases, there are still many examples where such heuristics lead to intolerable orderings. For these cases it is necessary to resort to a search of the space of possible conjunct orderings. Best-first search, together with the reduction theorems of Section 5.3.2, can significantly reduce the cost of this exploration, often to the extent that it is not significantly more expensive than the use of fallible heuristics such as cheapest-first.

Chapter 6

Controlling Backward Inference

6.1 Introduction

In the first chapter we argued that local domain-independent control procedures are not effective for solving difficult inference problems. The reason is that local properties of an inference step provide little clue as to whether or not the inference step is likely to lead to a solution, or what the cost will be of finding a solution by that path.

Consider a subset of the kinship axioms introduced in Chapter 1:

$$A: \ Brother(c,d) \Longrightarrow Sibling(c,d)$$
$$B: \ Parent(c,p) \wedge Parent(d,p) \wedge c \neq d \Longrightarrow Sibling(c,d)$$
$$C: \ Mother(c,p) \Longrightarrow Parent(c,p)$$
$$D: \ Father(c,p) \Longrightarrow Parent(c,p)$$

Suppose that the problem is to find a single solution to the goal $Sibling(Rob, s)$. The AND/OR graph of backward inference steps for this goal is shown in Figure 6.1. Initially, there are two different possible inference steps, labelled a and b in the graph. If the step b is chosen there are five additional possibilities, and so on.

Suppose the database contains many facts about brothers but few facts about mothers and fathers. In this case step a would be better than step b, *on the average*. For a particular database, and a particular problem, this advice might be wrong; b might be the better choice. But on the average, for problems of this form, step a is likely to lead to a solution more frequently and more quickly than step b. If there are roughly equal numbers of facts about brothers, mothers and fathers it would still be better, on the average, to try a and l_1 before trying b, since b produces a conjunction that is more expensive to solve than the single clause $Brother(Rob, s)$. On the other hand, if the database contains very few *Brother* facts, but many facts about mothers and fathers, it would be better, on the average, to do step b first.

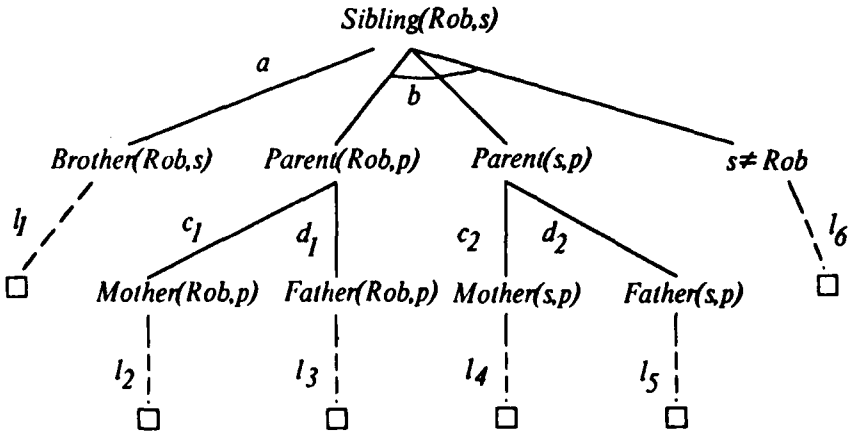


Figure 6.1: Backward AND/OR graph for the kinship problem. The letters used for the step labels indicate the rule used in the inference step. Steps labelled with the letter *l* are database lookup steps.

Local domain-independent strategies do not make any distinction between these different possibilities. Consider a method like choosing the step leading to the simplest subgoal. In this case step *a* would be chosen over step *b*, because *b* leads to a conjunction and *a* does not. The choice is the same regardless of the contents of the system's database, and regardless of the depth or complexity of the inference paths leading from *a* and *b*. Unfortunately, the simplicity or complexity of a goal says nothing about whether or not it is likely to be true, and provides only a crude lower bound on the cost of solving it. A simple goal may generate only complex subgoals, while a complex goal might have an immediate solution.

Thus, local properties of an inference step are not good indicators of the *promise* of the inference step. A good control decision requires an evaluation of the promise of the different possible steps. Such an evaluation must take into account both the likelihood that each inference action will lead to a solution, and the expected cost of finding a solution by that inference path. Both of these characteristics depend crucially on global information - what rules and facts are present in a system's database.

In this chapter, the problem of step selection for backward inference is considered, but the treatment is limited to problems where only a single answer is required. Normally, step selection would include selection of the ordering for conjunctive goals. However, this ordering problem is not considered here, although the issues involved are discussed in Section 6.5. For now, assume that the clauses in a

conjunction must be processed in the order given.

6.1.1 Approach

As discussed in Chapter 2, we are interested in finding the first step of the complete strategy with the lowest expected cost. It is impossible to precisely determine the cheapest complete strategy without a search of the inference space that is more expensive than finding a solution to the problem. However, using an averaging assumption like that used in the case study on conjunct ordering, it is possible to give a statistical estimate of the promise of each inference step. Using this information, the expected cost and probability of success for different strategies can be computed. This information can then be used to find the complete strategy that will, on the average, be the best single strategy for all problems of the same form.

These calculations rely on an assumption about independent inference steps. Two inference steps are *independent* if neither requires that the other be performed first. In our example, the steps a and b are independent but the steps a and l_1 are not, since l_1 requires that a be performed first. The assumption is that the cost of an inference step, and the likelihood that it will contribute an answer to the problem are not affected by the performance of other independent inference steps. Usually this is a correct assumption, but it can be violated by redundancy in the search space or by the use of caching when the inference space is a graph rather than a tree. The independence assumption is considered in more detail in the discussion of Section 6.5.

In computing the expected cost and probability of success for strategies we will make use of information about the topology of the inference space for all goals involving the same relation. We will also make use of the expected cost of each step in the inference space, the likelihood that each inference step will be possible for an arbitrary query of the given form, and the likelihood that each database lookup step will actually yield a solution to the problem.

It might appear that this approach requires a complete search of the inference space. Yes, and no. The space must be searched once for each goal form, but not for each individual goal. Thus, the savings come about by amortizing this control search over many similar problems. Also, as was the case in ordering conjuncts we only search each branch in a conjunction once rather than once for every answer produced, as required in actual problem solving.

6.1.2 Organization

In Section 6.2 we show how to compute the probability that a goal expression will match an axiom or a fact in the database. In Section 6.3, this information is used

to compute the probability and expected cost for an arbitrary strategy. In Section 6.4, reduction theorems are developed that indicate which of the many different strategies need to be considered for a given problem. Finally, Section 6.5 is about the possibilities and difficulties involved in extending this analysis to bidirectional inference, and to problems requiring multiple answers.

6.2 Computing Local Probability

Consider a particular subgoal of the form $P = R(\vec{x})|_b$ where b is some set of bindings of the set of variables v which is some subset of the variables in $\vec{x}$. The local probability of an inference step is defined to be the probability that the inference step will succeed. For a backward inference step, this is the probability that the conclusion of the particular rule of the inference step will match the given goal espression. For a database lookup step, this is the probability that a fact in the database will match the given goal expression.

6.2.1 Inference Steps

Given an implication (or more generally a disjunction) of the form $\phi \Longrightarrow R(\vec{\alpha})$, we want to know the probability that a goal expression $R(\vec{x})|_b$ will match the consequent of this implication. If $\vec{\alpha}$ contains variables in every position bound by b the probability will be 1.

$$\forall x_i(x_i \in v \Longrightarrow Variable(\alpha_i)) \Longrightarrow P(Unify(R(\vec{x})|_b, R(\vec{\alpha}))) = 1$$

If not, the probability that the constants are the same must be considered. This is simply the inverse of the product of all the domain sizes involved.

$$P(Unify(R(\vec{x})|_b, R(\vec{\alpha}))) = \prod_{\{i:x_i \in v \wedge \neg Variable(\alpha_i)\}} \frac{1}{\|Domain(R,i)\|} \qquad (6.1)$$

6.2.2 Database Lookup Actions

Calculating the probability that there is a fact in the database matching a given goal expression is more difficult than the calculations for implications. The difference is that we are not calculating the probability that an individual fact will match a given goal expression, but rather the probability that at least one fact in the database will match the goal expression. The latter is not a simple function of the former because each fact in the database is unique.

To compute the probability that there are k or more facts in the database matching a given goal expression $R(\vec{x})|_b$, we need to know three things,

1. the number of facts in the database having the relation R,
2. the number of different possible ways in which the variables v in b could be bound, and
3. the average number of solutions theoretically possible for $R(\vec{x})|_b$ over the class of all legal bindings b for the variables v.

The first quantity, the number of facts in the database having the relation R, is given by

$$m = \| \{f : Match(f, R(\vec{x}))\} \| ,$$

where

$$Match(f, g) \equiv InDataBase(f) \wedge Unify(f, g).$$

This can be computed by simply counting the facts in the database.

Let D_v be the cross product of the domains for the arguments of R bound by b.

$$D_v = \bigtimes_{\{i:x_i \in v\}} Domain(R, i)$$

The second quantity, the number of different possible ways of binding the variables v, is just the cardinality of the domain D_v,

$$g = \|D_v\| = \prod_{\{i:x_i \in v\}} \|Domain(R, i)\| .$$

To compute this, we need to know the sizes of the domains for the arguments of the relation R.

The third quantity can be expressed using the terminology for averages and number of solutions developed in the previous chapter.

$$h = \underset{b \in D_v}{\text{Avg}}(Numsol(R(\vec{x})|_b)) = AvgNumsol(R(\vec{x})|_b, b \in D_v)$$

This quantity is always bounded by the cardinality of the cross product of the domains for the remaining unbound arguments of R.

$$h \leq \|D_{\bar{v}}\| = \prod_{\{i:x_i \notin v\}} \|Domain(R, i)\|$$

As in the case of conjunct ordering (Section 5.2), h can often be calculated more exactly using readily available information. For example, we might have knowledge that a person has two parents, or that a particular relation is functional when certain of its arguments are bound. In Section 6.2.3 this information of this sort is used to compute h for one of the subgoals in the kinship problem.

Theorem 6.1 *The probability of finding k or more solutions ($k \leq m$) in the database for a query $R(\vec{x})|_b$ is*[1]

$$P(\exists_{\geq k} f : Match(f, R(\vec{x})|_b)) = 1 - \sum_{i=0}^{k-1} \frac{\binom{h}{i}\binom{(g-1)h}{m-i}}{\binom{gh}{m}} . \tag{6.2}$$

Proof *The probability that there are k or more facts in the database matching a given goal expression is one minus the probability that there are fewer than k such propositions in the database ($k \leq m$ is assumed throughout the proof).*

$$\begin{aligned} &P(\exists_{\geq k} f : Match(f, R(\vec{x})|_b)) \\ &\quad = 1 - P(\exists_{<k} f : Match(f, R(\vec{x})|_b)) \\ &\quad = 1 - \sum_{i=0}^{k-1} P(\exists_k f : Match(f, R(\vec{x})|_b)) \end{aligned}$$

The probability that there are exactly k solutions to $R(\vec{x})|_b$ is isomorphic to the following problem.

> *Given g egg cartons, each of which has h holes in it, and m marbles distributed randomly over all g times h holes (no hole can hold more than one marble) what is the probability that a given egg carton will contain exactly k marbles?*

An illustration of the problem is shown in Figure 6.2.

There are

$$\binom{h}{k}$$

different ways of placing k marbles into the h holes of the one particular egg carton we are interested in. Likewise, there are

$$\binom{(g-1)h}{m-k}$$

different ways of placing the remaining $m-k$ marbles into the remaining $g-1$ egg cartons. As before, there are

$$\binom{gh}{m}$$

[1] Here we assume that the set of bindings b for the goal form $R(\vec{x})|_b$ lie within the corresponding domains for the relation R, i.e. $b \in D_v$. This assumption is often true for subgoals generated internally. When this is not true, the total probability must be multiplied by the probability that variable bindings lie within the appropriate domains, as we did for conjunct ordering in Section 5.2.

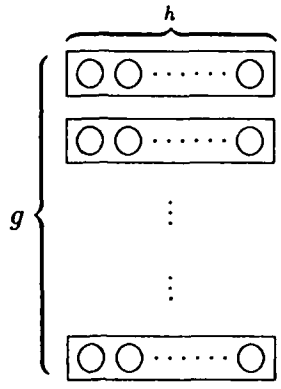


Figure 6.2: Egg carton model

different ways of placing m marbles into all of the cartons. Thus the probability that there are exactly k marbles in the carton we have chosen is

$$\frac{\binom{h}{k}\binom{(g-1)h}{m-k}}{\binom{gh}{m}}.$$

So, the probability that there are exactly k solutions in the database matching $R(\vec{x})|_b$ for arbitrary $b \in v$ is

$$P(\exists_k f : Match(f, R(\vec{x})|_b)) = \frac{\binom{h}{k}\binom{(g-1)h}{m-k}}{\binom{gh}{m}}$$

and the probability that there are at least k solutions (for $k \leq m$) is

$$P(\exists_{\geq k} f : Match(f, R(\vec{x})|_b)) = 1 - \sum_{i=0}^{k-1} \frac{\binom{h}{i}\binom{(g-1)h}{m-i}}{\binom{gh}{m}}.$$

□

For $k = 1$ the above equation reduces to

$$P(\exists f : Match(f, R(\vec{x})|_b)) = 1 - \frac{\binom{(g-1)h}{m}}{\binom{gh}{m}}. \tag{6.3}$$

There are several useful special cases of this relationship. When $m = 1$ (i.e. only one fact in the database has the desired relation), the probability reduces to

$$P(\exists f : Match(f, R(\vec{x})|_b)) = 1 - \frac{\binom{(g-1)h}{1}}{\binom{gh}{1}} = 1 - \frac{g-1}{g} = \frac{1}{g}.$$

Another useful special case is when

$$h = AvgNumsol(R(\vec{x})|_b, b \in D_v) = 1.$$

In other words, there is only one hole in each egg carton. This would be the case if R is functional when the variables v are bound. In this situation there can be only one answer and the complicated expression given above reduces to

$$P(\exists f : Match(f, R(\vec{x})|_b)) = 1 - \frac{\binom{g-1}{m}}{\binom{g}{m}} = 1 - \frac{g-m}{g} = \frac{m}{g}. \tag{6.4}$$

This is as we would expect, since there are precisely m chances of having a marble put in one particular egg carton. Likewise, this answer holds for the case in which all of the variables to R are bound, making $R(\vec{x})|_b$ a ground clause.

Finally, if none of the variables to R are bound (i.e. v is empty) the probability of k or more answers for $R(\vec{x})$ is 1 if $k \leq m$ and zero otherwise

$$P(\exists_{\geq k} f : Match(f, R(\vec{x}))) = \begin{cases} 1, & \text{if } k \leq m; \\ 0, & \text{otherwise.} \end{cases}$$

6.2.3 Example

Now let us apply these equations to the kinship example, shown in Figure 6.1, with a database consisting of the facts

$$\begin{aligned} &Brother(Rob, Larry) \\ &Father(Rob, Bill) \\ &Mother(Rob, Terry) \\ &Father(Pat, Bill) \\ &Father(Larry, Bill)\ . \end{aligned}$$

Since none of the rules for this problem have any of the arguments bound in their conclusions, all of the local probabilities for inference steps are one. In other words, the rules in the kinship problem will always apply independent of which variables in the query are bound.

We are left with calculating the local probability for only the database lookup actions. For the goal expression, $Brother(Rob, s)$, we note that there is one $Brother$ fact available in the database, so $m = 1$. The domain for both arguments of this relation is the set of persons, which for our example has been limited to only five individuals.

$$P(\exists f : Match(f, \text{“}Brother(x, s)|_x\text{”})) = \frac{1}{\|Domain(Brother, 1)\|} = \frac{1}{5} = .2$$

By similar reasoning, the probabilities for the steps l_2 and l_4 (looking up solutions to clauses containing the *Mother* relation) are all .2 .

$$P(\exists f : Match(f, \text{“}Mother(x,p)|_x\text{”})) = \frac{1}{\|Domain(Mother, 1)\|} = \frac{1}{5} = .2$$

The more interesting calculations are those for goal expressions containing the *Father* relation. Consider step l_3, the lookup step for the clause $Father(Rob, p)$. There are three fatherhood facts in the database so $m = 3$. Since $Father(x, p)$ is functional when x is bound we can use the restricted form for functional expressions (equation 6.4).

$$P(\exists f : Match(f, \text{“}Father(x,p)|_x\text{”})) = \frac{m}{g} = \frac{3}{5} = .6$$

For the step l_5 the functional relationship does not hold. In this case we must use the full power of equation (6.2). Again, $m = 3$, $g = 5$, and $h = 5$.

$$P(\exists_0 f : Match(f, \text{“}Father(x,p)|_p\text{”})) = \frac{\binom{(g-1)h}{m}}{\binom{gh}{m}} = \frac{\binom{4*5}{3}}{\binom{5*5}{3}} = .4957$$

So,

$$P(\exists f : Match(f, \text{“}Father(x,p)|_p\text{”})) = 1 - .4957 = .5043$$

For these cases there is a non-zero chance that a second or third answer can be found, so we must also calculate these probabilities.

$$P(\exists_1 f : Match(f, \text{“}Father(x,p)|_p\text{”})) = \frac{\binom{h}{k}\binom{(g-1)h}{m-k}}{\binom{gh}{m}} = \frac{\binom{5}{1}\binom{20}{2}}{\binom{25}{3}} = .4130$$

$$P(\exists_{\geq 2} f : Match(f, \text{“}Father(x,p)|_p\text{”})) = 1 - .4957 - .4130 = .0913$$

$$P(\exists_2 f : Match(f, \text{“}Father(x,p)|_p\text{”})) = \frac{\binom{5}{2}\binom{20}{1}}{\binom{25}{3}} = .0870$$

$$P(\exists_{\geq 3} f : Match(f, \text{“}Father(x,p)|_p\text{”})) = 1 - .4957 - .4130 - .0870 = .0043$$

This data is summarized in Table 6.1.

Finally, consider the clause $x \neq y$ where both x and y are bound. Since the universe contains only five individuals there are twenty of these facts implicitly available in the database ($m = 20$). The product of the domain sizes g is 25 so the probability will be $\frac{20}{25}$ or .8 .[2]

[2]For a large database this probability would approach one. In general, to get a more realistic estimate for a clause like this we would have to make use of information about previous conjuncts that bind the variables involved. For this example, the variables will be restricted to the domain of people by previous conjuncts.

k	$P(\exists_k)$	$P(\exists_{\geq k})$	$\frac{P(\exists_{>k})}{P(\exists_{\geq k-1})}$
0	.4957	1	
1	.4130	.5043	.5043
2	.0870	.0913	.1811
3	.0043	.0043	.0476

Table 6.1: Probability of k or more solutions to $Father(s, p)$ when p is bound

6.3 Computing Expected Cost for a Strategy

In the previous section we showed how to compute the probability that the right hand side of a rule would match a given goal form and the probability that a given goal form would match some fact in the database. Given this information, the probability of success and expected cost for inference steps and strategies can be computed.

It is important, here, to distinguish between two different probabilities:

1. *Local probability*, $L(s)$ – the probability that an individual step will succeed.

2. *Global probability*, $P(s)$ – the probability that a step or strategy will actually solve the overall problem.

The global probability for a step includes the local probability that the step can be performed. The global probability can be expressed as the product of the local probability with another term that represents the probability of solving the problem given that the action is possible.

$$P(s) = L(s)\rho(s)$$

For example, the probability that we can get a certain item at Gumps is the product of the probabilities of being able to get to Gumps and that Gumps has the desired item.

For individual inference steps the global probability is zero, $P(s) = \rho(s) = 0$. For example, in our simple kinship problem, reducing the goal $Parent(Rob, p)$ to $Father(Rob, p)$ cannot, by itself, solve the problem. For database lookup steps there are two possibilities. If the goal set has more than one conjunct, the global probability will also be zero, since finding a solution to only one conjunct does not provide a solution to the problem. In other words, the global probability is zero for all individual inference steps except those corresponding to the leaf branches of the OR-search space. For example, in the kinship problem, looking up the solutions

Figure 6.3: Combining independent strategies

to $Father(Rob, p)$ does not solve the problem unless the other branches to the conjunction have already been solved.

For database lookup steps that are leaf branches in the OR-space the global probability is not zero. For these branches there is only a single conjunct remaining, so any solution found to this conjunct will provide a solution to the overall problem. Thus, $\rho(s) = 1$, giving $P(s) = L(s)$. For the kinship problem $\rho(l_1) = \rho(l_6) = 1$ giving $P(l_1) = L(l_1) = .2$ and $P(l_6) = L(l_6) = .8$.[3] $P(s) = \rho(s) = 0$ for all other steps.

We now turn to the task of computing the expected cost of a strategy given the local probability $L(s)$, the global probability $P(s)$, and the expected cost $E(s)$ for each of its steps. First consider the case of computing expected cost for a strategy xy where the steps x and y are *independent*, as illustrated in Figure 6.3. (Two steps are independent if neither depends upon the ability to perform the other.) In this case, x will always be performed, and y will be performed if x fails to solve the problem.

$$E(xy) = E(x) + (1 - P(x))E(y) \tag{6.5}$$

Likewise, the probability that xy will succeed is

$$P(xy) = P(x) + (1 - P(x))P(y).$$

In fact, these equations hold for any completely independent pair of strategies x and y (not just single steps). (Two strategies are independent if no step in either strategy, depends on the ability to perform any step in the other strategy.)

Alternatively, suppose that a step y depends upon the success of a step x, i.e. y cannot be performed unless x is performed. This situation is illustrated in Figure 6.4. As before, x will always be performed. But in this case, if x is not possible, y will not be performed. It is only when x is possible, but does not solve the problem that y will be performed.

$$\begin{aligned} E(xy) &= E(x) + L(x)(1 - \rho(x))E(y) \\ &= E(x) + (L(x) - P(x))E(y) \end{aligned} \tag{6.6}$$

[3]There is more than one way to think about the values for the branches l_1 and l_6. We choose to let $\rho(s) = 1$ for these two steps and let $L(s)$ be .2 and .8 respectively. The reverse is just as good. What really matters is the value of $P(s)$, which is the same for either assignment.

Figure 6.4: Combining dependent strategies

Similarly, the probability of success for xy is

$$\begin{aligned} P(xy) &= P(x) + L(x)(1-\rho(x))P(y) \\ &= P(x) + (L(x) - P(x))P(y) \ . \end{aligned} \tag{6.7}$$

In contrast to the independent case, these formulae do not hold for arbitrary strategies x and y. There are two restrictions,

1. the strategy x must be a single step, and
2. the strategy y must be *singly-rooted*, i.e. all other steps in the strategy must depend upon the first step, and must not depend on any steps not in the strategy.

The first restriction is necessary because, if x is not a single step, then for each step in x it is necessary to determine whether or not y depends upon the ability to perform that step. The second restriction is necessary because if y has more than one root, some portions of y will depend upon x while other portions will not.

Let $P_0(s,t)$ refer to the probability that the strategy s fails in such a way that the singly-rooted strategy t is still possible. In other words, P_0 refers to the probability that s fails to find an answer to the problem, but that all of the steps in s that t depends on, were possible. Using this notion, expected cost and probability calculations can be generalized to arbitrary strategies.

Theorem 6.2 *If t is a singly rooted strategy,*

$$\begin{aligned} E(st) &= E(s) + P_0(s,t)E(t) \ , \\ P(st) &= P(s) + P_0(s,t)P(t) \ . \end{aligned}$$

Proof *The strategy s will be attempted in every case. The expected cost of this is $E(s)$. If s solves the problem, t will not be executed. If s fails to solve the problem, the strategy t will be attempted, but only if all of the steps that t depends on were possible. The probability of this is $P_0(s,t)$. Thus, in that fraction of the cases there will be the additional cost $E(t)$. The expected cost is*

$$E(st) = E(s) + P_0(s,t)E(t) \ .$$

The argument for the probability equation is similar. □

Corollary 6.3 *If an arbitrary strategy s has n individual steps, $s_1, \ldots, s_n$, the expected cost for the strategy will be the sum over all steps of the expected cost of the individual step times the probability that the preceding steps fail in such a way that the step will still be possible. Likewise for the probability of an arbitrary strategy.*

$$E(s) = \sum_{i=1}^{n} P_0(s_{1,i-1}, s_i) E(s_i)$$

$$P(s) = \sum_{i=1}^{n} P_0(s_{1,i-1}, s_i) P(s_i) .$$

In order to use these results, P_0 must be computed. First, consider two special cases. If s and t are independent, whenever s fails, t will be possible.

$$P_0(s,t) = 1 - P(s)$$

Alternatively, if t depends upon every step in s, every step in s must be possible for t to be possible.

$$P_0(s,t) = \prod_{x \in s} P_0(x,t) = \prod_{x \in s} (L(x) - P(x))$$

More generally P_0 can be computed by breaking up a strategy into its constituent parts according to the following theorem.

Theorem 6.4 *Let s be divided into two subsequences s' and s'' such that for every $x \in s'$ and $y \in s''$ that are both independent of t, x is independent of y (in other words, each branch that is independent of t must be wholly contained in either s' or s''). Then*

$$P_0(s,t) = P_0(s',t) P_0(s'',t) .$$

Proof *Let $s_{I(t)}$ refer to the subsequence of steps in s that are independent of t. Similarly, let $s_{D(t)}$ refer to the subsequence of steps in s that t depends on. If the strategy s is to fail in such a way as to leave t possible, each step in s must fail in such a way as to leave t possible. For the dependent steps, $s_{D(t)}$, each step must be possible but must fail to produce a solution to the problem. The probability of this is*

$$P_0(s_{D(t)}, t) = \prod_{x \in s_{D(t)}} P_0(x,t) = \prod_{x \in s_{D(t)}} (L(x) - P(x)) .$$

Given that the dependent steps are possible, but fail to produce an answer, the independent steps, $s_{I(t)}$, must fail, either by not being possible, or by failing to produce an answer to the problem. The probability of this is

$$P_0(s_{I(t)}, t) = 1 - P(s_{I(t)})$$

The probability that both of these events will take place is therefore just the product of these two terms.

$$P_0(s, t) = P_0(s_{D(t)}, t) P_0(s_{I(t)}, t)$$

Now since $P_0(s_{D(t)}, t)$ is a product over the individual steps,

$$P_0(s_{D(t)}, t) = P_0(s'_{D(t)}, t) P_0(s''_{D(t)}, t) .$$

Likewise, by the independence condition of the theorem, $S'_{I(t)}$ and $S''_{I(t)}$ are independent, so,

$$P_0(s_{I(t)}, t) = P_0(s'_{I(t)}, t) P_0(s''_{I(t)}, t) .$$

Therefore,

$$\begin{aligned} P_0(s, t) &= P_0(s_{D(t)}, t) P_0(s_{I(t)}, t) \\ &= P_0(s'_{D(t)}, t) P_0(s''_{D(t)}, t) P_0(s'_{I(t)}, t) P_0(s''_{I(t)}, t) \\ &= P_0(s', t) P_0(s'', t) . \end{aligned}$$

□

One of the most useful ways to do this computation is to break up the strategy s into its independent and dependent steps.

Corollary 6.5

$$\begin{aligned} P_0(s, t) &= P_0(s_{I(t)}, t) P_0(s_{D(t)}, t) \\ &= (1 - P(s_{I(t)})) \prod_{x \in s_{D(t)}} (L(x) - P(x)) \end{aligned}$$

The equations for computing expected cost and probability are summarized in Figure 6.5.

6.3.1 Example

Using the above equations, together with the local probability data computed in Section 6.2, the expected costs and probability of success for strategies in the kinship example can be computed. In Figure 6.6, a portion of OR-space for this problem is shown. Only the branches for one of the possible conjunct orderings is shown. The

General Formulae

$$E(st) = E(s) + P_0(s,t)E(t)$$

$$P(st) = P(s) + P_0(s,t)P(t)$$

$$P_0(s,t) = P_0(s',t)P_0(s'',t)$$

provided $s'_{I(t)}$ and $s''_{I(t)}$ are independent

Special Cases

$$P_0(s_{I(t)},t) = 1 - P(s)$$

$$P_0(s_{D(t)},t) = \prod_{x \in s_{D(t)}} (L(x) - P(x))$$

Figure 6.5: Expected cost and probability combination for strategies (t must be singly-rooted).

local probability numbers computed in Section 6.2 for each branch are shown along side the branch.[4]

For simplicity assume that the cost of each inference step is one and the cost of looking up each answer in the database is .1 . For the partial strategy $l_4 l_6$ the global probability and expected cost are

$$\begin{aligned} P(l_4 l_6) &= P(l_4) + P_0(l_4, l_6)P(l_6) \\ &= P(l_4) + L(l_4)P(l_6) \\ &= 0 + (.2)(.8) \\ &= .16 \\ E(l_4 l_6) &= E(l_4) + P_0(l_4, l_6)E(l_6) \\ &= E(l_4) + L(l_4)E(l_6) \\ &= .1 + (.2)(.1) \\ &= .12 \end{aligned}$$

The calculations for several partial strategies are summarized in Table 6.2. One of the more interesting steps is the calculation for the strategy $c_1 l_2 d_2^* c_2^*$ given the global

[4]Only the first branch for the database lookup steps l_5 and l'_5 are shown. In reality the pairs $l_5 l''_6$ and $l'_5 l'''_6$ should be duplicated two more times each, since there is non zero probability that the conjunct $Father(s,P)$ will have two or even three solutions. These additional branches would have the probabilities .0913 and .0043 as we calculated in Section 6.2 and summarized in Table 6.1.

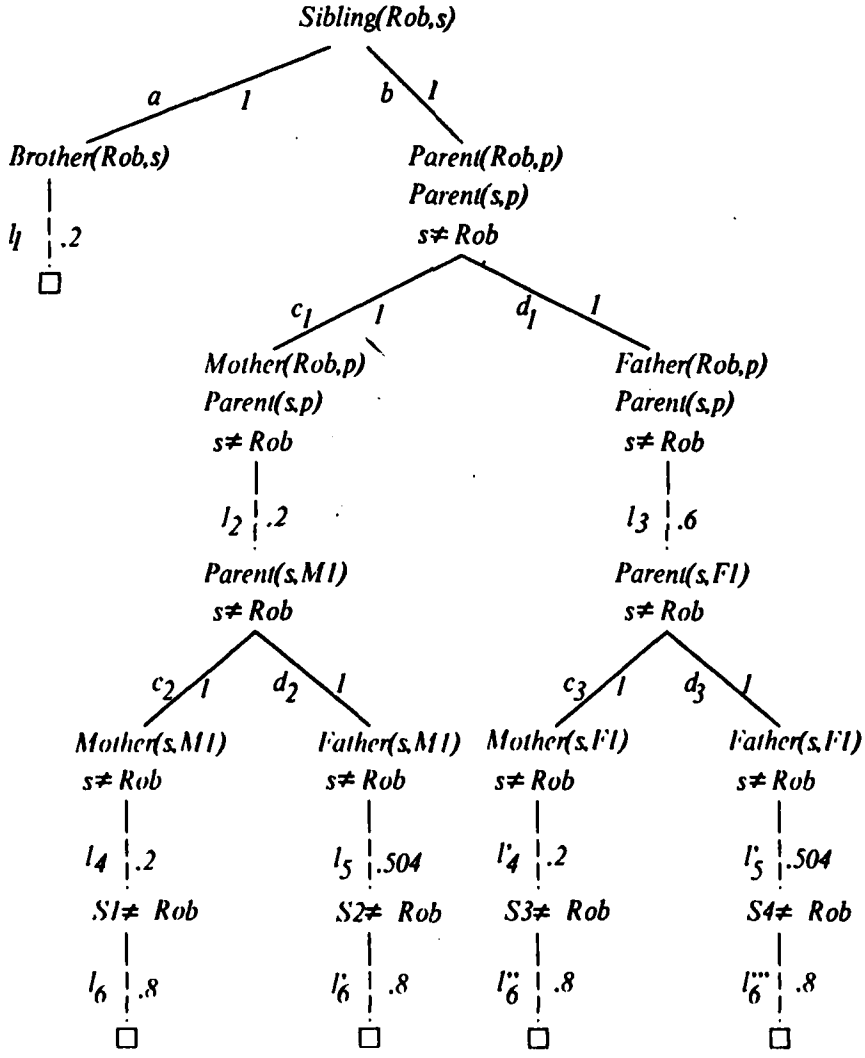


Figure 6.6: OR version of the kinship inference space with local probability evaluations. Goals containing more than one expression are conjunctions.

probability and expected cost for $c_1 l_2 d_2^*$ and c_2^*. For this case the general expected cost and probability equations in Figure 6.5 are required. First, P_0 is calculated.

$$\begin{aligned} P_0(c_1 l_2 d_2^*, c_2^*) &= L(c_1)L(l_2)(1 - P(d_2^*)) \\ &= 1(.2)(1 - .4034) \\ &= .119 \end{aligned}$$

Using this information probability and expected cost can be computed.

$$\begin{aligned} P(c_1 l_2 d_2^* c_2^*) &= P(c_1 l_2 d_2^*) + P_0(c_1 l_2 d_2^*, c_2^*)P(c_2^*) \\ &= .0807 + .119(.16) \\ &= .1 \\ E(c_1 l_2 d_2^* c_2^*) &= E(c_1 l_2 d_2^*) + P_0(c_1 l_2 d_2^*, c_2^*)E(c_2^*) \\ &= 1.33 + .119(1.12) \\ &= 1.46 \end{aligned}$$

In a similar manner, the global probability and expected cost can be computed for other partial strategies. The calculations for the complete strategy

$$a^* b d_1 l_3 d_3^* c_3^* c_1 l_2 d_2^* c_2^*$$

are summarized in Table 6.2. The total probability of success is .5 and the expected cost is 4.5. Thus, to solve this problem using this strategy would, on the average, require the equivalent of four inference steps of effort.

6.4 Strategy Reduction Theorems

Having shown how to calculate expected cost and probability of success for different strategies in a search space it is now time to return to the problem of choosing the optimal strategy. Given the information in the previous sections, it is possible to enumerate every complete strategy, calculate the expected cost for each and compare to find the best one. (Every complete strategy has the same probability of success). Although this is theoretically possible it is quite intractable. Even if we only consider the restricted set of inference actions in the kinship space of Figure 6.6, the number of different possible complete strategies (legal ones) is 880,308,000. Fortunately, this combinatoric process is entirely unnecessary. In the next two subsections reduction theorems are developed that tell us which of the 10^9 strategies we need to consider in order to find the optimal one. In the case of our simple kinship problem the reduction theorems will allow us to incrementally build up the optimal strategy. In fact, no search will be necessary for inference spaces that are trees (as opposed to graphs). For spaces that are graphs, the number of possible strategies that must be considered will also be substantially reduced.

Strategy s	$P(s)$	$E(s)$	$U(s) = \frac{P(s)}{E(s)}$
$l_4 l_6$	.16	.12	1.33
$c_2^* = c_2 l_4 l_6$	.16	1.12	.143
$l_5 l_6''$	.4034	.15	2.69
$d_2^* = d_2 l_5 l_6''$	.4034	1.15	.35
$l_2 d_2^*$	.0807	.33	.245
$c_1 l_2 d_2^*$	.0807	1.33	.078
$c_1 l_2 d_2^* c_2^*$	.1	1.46	.068
$l_4' l_6'''$	.16	.12	.11
$c_3^* = c_3 l_4' l_6'''$	.16	1.12	1.33
$l_5' l_6''''$	.4034	.15	2.69
$d_3^* = d_3 l_5' l_6''''$	.4034	1.15	.35
$l_3 d_3^*$	.242	.79	.306
$d_1 l_3 d_3^*$	.242	1.79	.135
$d_1 l_3 d_3^* c_3^*$	.3	2.19	.137
$b d_1 l_3 d_3^* c_3^*$	.3	3.19	.094
$a^* = a l_1$	.2	1.1	.182
$a^* b d_1 l_3 d_3^* c_3^*$	.44	3.65	.121
$a^* b d_1 l_3 d_3^* c_3^* c_1 l_2 d_2^* c_2^*$	.5	4.47	.112

Table 6.2: Expected cost, probability of success and utility data for partial strategies in the kinship problem.

6.4.1 The General Optimality Theorem

The *utility* of a strategy is defined to be the ratio of the probability of success to the expected cost for the strategy

$$U(s) = \frac{P(s)}{E(s)}. \tag{6.8}$$

In Table 6.2, the utility is included along with the expected cost and probability of success for the partial strategies considered in Section 6.3.1. In the case of inference problems the utility for any partial strategy that does not include a terminal step will be zero. This is because the probability of success for such strategies is zero.

Let a be any inference action in the search space and let s refer to the the set of all inference steps that are successors of a. The *best* strategy, t, for the step a, is defined as the strategy on s that begins with a and has the highest utility. The utility $U(t)$ can be thought of as the *promise* of the inference step a, i.e. the most *good* that can possibly come from performing the step a. This is the number used to decide among different steps.

Conjecture 6.6 *The optimal strategy will consist of the maximal best strategies ordered by decreasing utility.*

This conjecture is a generalization of a theorem proven by Simon and Kadane [SK75] and applies to arbitrary partially ordered strategy spaces (i.e. graphs as well as trees). Simon and Kadane prove the result for graphs involving expected cost and global probability, but do not consider local probability factors. In other words, local probability factors are assumed to be one. It appears that the proof carries through identically for the more general case, although we have not verified all of the details. Several of the supporting lemmas required for this proof have been shown to hold.

In general, the best strategy for a step a may consist of only the action a, or it may contain many or all of the successors s of a. However, if the best strategy for a contains a step b it will also include the best strategy for b. A *maximal* best strategy for a search space is one that is not contained in any other best strategy.

At first, this result may appear rather weak because it does not tell us how to go about constructing the maximally best strategies for a search space. However, this result does allow the incremental construction of the optimal strategy. Starting at the leaf nodes of a search space, one can work upward through the space constructing larger and larger best strategies. Once a best strategy block is formed, it can be treated as an individual unit, and the steps for that block need not be considered separately again.

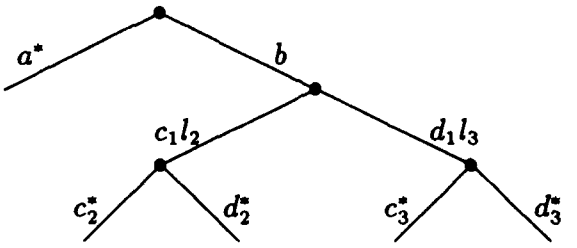


Figure 6.7: Reduced strategy space for the kinship problem

A second thing to note is that for the case of inference, a best strategy must always include at least one terminal node. The global probability is zero for non-terminal steps, so any strategy that does not contain a terminal will have probability and utility of zero. Thus, the best strategy must contain a terminal. This observation, coupled with the theorem above, allows drastic reduction of the strategy space for the kinship example. For example, the best strategy beginning with c_2 must contain a terminal and will, therefore, be $c_2^* = c_2 l_4 l_6$. Likewise for d_2, c_3, and d_3 respectively. The best strategy from a is $a l_1$. The best strategy for c_1 must also contain a terminal and must therefore begin $c_1 l_2$. Likewise, for $d_1 l_3$. This reduced strategy space is shown in Figure 6.7. Already, the number of indivisible steps that must be considered is reduced from 19 to only 8, and the entire set of strategies possible has been reduced from 10^9 to only 640.

By working upwards through the tree, the best strategies for the example can be discovered with minimal effort. Once this is done, the theorem states that the optimal ordering will consist of these blocks ordered by decreasing utility.

6.4.2 Garey's Reduction Theorems

The optimality theorem of Simon and Kadane can be used to significantly reduce the amount of search involved in finding an optimal strategy for a problem. For the case in which the search space is a tree, an even more powerful reduction theorem applies.

Conjecture 6.7 *Given a step x, let y be the successor of x having maximum utility $U(y)$. Suppose that $U(y) > U(x)$, and all other descendants of x have utility less than that of y. The step y will immediately follow the step x in any optimal strategy. Furthermore, since all successors of x must necessarily follow xy in the optimal strategy, the strategy x can be replaced by xy and the strategy y can be deleted (i.e. all of y's successors become successors of xy).*

This conjecture is a generalization of a theorem proven by Garey [Gar73]. Garey only proves this result for trees involving expected cost and global probability. In other words, the local probability factors required here are assumed to be one. It appears that Garey's results also hold for the more general case, although we have not verified all of the details of the proof. Following Simon and Kadane [SK75], we refer to a strategy xy satisfying the conditions of this theorem as an *indivisible block*. This reduction theorem can also be used when x and any of its successors are indivisible blocks rather than individual actions. The utility of the indivisible blocks is computed using (6.8), and the equations in Figure 6.5.

Garey goes on to show that, for trees, if all *maximal* indivisible blocks are formed using Theorem 6.7, the optimal strategy will consist of these blocks ordered by decreasing utility. Finding the optimal ordering for trees is therefore a relatively simple process.

1. Partition the strategy space into the set of maximal indivisible blocks.
2. Order the blocks by decreasing utility.

The first step of this process can be performed in an efficient manner using a marking algorithm that begins at the leaf nodes of the search space.

Algorithm 6.8

1. *Mark all unmarked leaf actions in the tree.*
2. *Choose an unmarked action x all of whose successors are marked. Let y be the successor having maximal utility.*
3. *If $U(y) \leq U(x)$ mark x and return to step 2. Otherwise, replace x by xy, remove y and let all of y's successors be successors of xy. Return to step 1.*

Notice that the expensive condition in Theorem 6.7, of verifying that all other descendants have lower utility, is automatically satisfied by the bottom up marking process in this algorithm.

6.4.3 Example

Given the reduction theorems of the previous sections, the optimal strategy for the kinship example can be found, by making use of the probability, expected cost, and utility computations given earlier in Table 6.2. Initially, the terminal branches l_1, l_6, l'_6, l''_6 and l'''_6 are marked by the algorithm. An action is chosen whose successors are marked. The actions a, l_4, l_5, l'_4 and l'_5 all have this characteristic. Start with the action a. It has utility of zero, while its only descendant l_1 has non-zero utility.

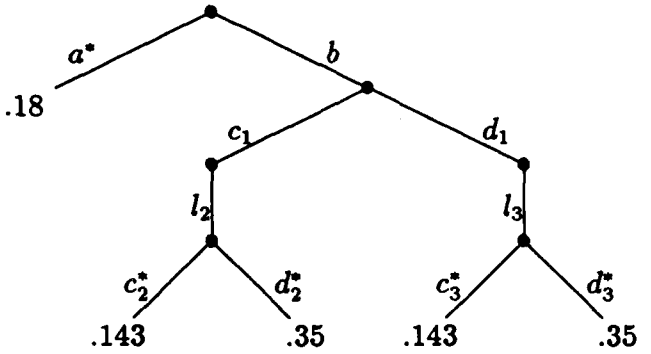


Figure 6.8: First reduction for the kinship problem. Utilities are shown for those steps where it is non-zero.

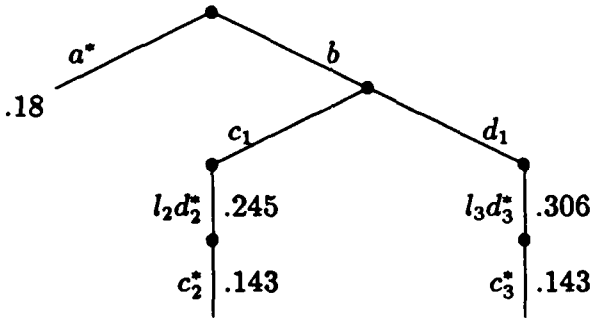


Figure 6.9: Second reduction for the kinship problem. Utilities are shown for those steps where it is non-zero.

Therefore, al_1 forms an indivisible block and a is replaced with al_1 and l_1 is deleted from the graph. Since al_1 has no descendants, it is a leaf node and is marked. In similar fashion we form and mark the indivisible blocks l_4l_6, l_5l_6', $l_4'l_6''$ and $l_5'l_6'''$. Now the steps c_2, d_2, c_3 and d_3 all have marked successors. The utilities of these steps is also zero, while the utilities of their successors is non-zero. Therefore, c_2^*, d_2^*, c_3^* and d_3^* form indivisible blocks. The utility for these blocks is computed and shown in Table 6.2 and the reduced strategy graph is shown in Figure 6.8.

The steps a^*, c_2^*, d_2^*, c_3^* and d_3^* are now marked so we consider the steps l_2 and l_3. From the reduced space we see that $U(d_2^*) > U(c_2^*) > U(l_2) = 0$. The indivisible block $l_2d_2^*$ is therefore formed by the algorithm. Likewise for the block $l_3d_3^*$. The reduced space is shown in Figure 6.9.

Now, $U(l_2d_2^*) > U(c_2^*)$, so $l_2d_2^*$ is marked and the process continues. Likewise,

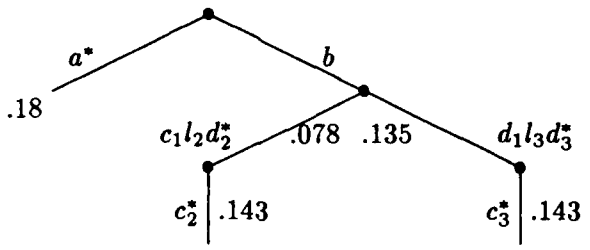


Figure 6.10: Third reduction for the kinship problem. Utilities are shown for those steps where it is non-zero.

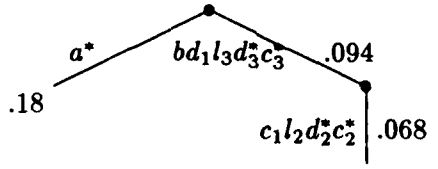


Figure 6.11: Final reduction for the kinship problem. Utilities are shown for those steps where it is non-zero.

for $l_3 d_3^*$. Moving up a level in the tree, we consider c_1 and d_1. $U(l_2 d_2^*) > U(c_1)$ and $U(l_3 d_3^*) > U(d_1)$, so the indivisible blocks $c_1 l_2 d_2^*$ and $d_1 l_3 d_3^*$ are formed. The reduced space is shown in Figure 6.10.

Having performed this step the only marked branches are a^*, c_2^* and c_3^*. As a result, $c_1 l_2 d_2^*$ and $d_1 l_3 d_3^*$ are considered next. $U(c_2^*) > U(c_1 l_2 d_2^*)$ and $U(c_3^*) > U(d_1 l_3 d_3^*)$, so the indivisible blocks $c_1 l_2 d_2^* c_2^*$ and $d_1 l_3 d_3^* c_3^*$ are formed.

When the entire reduction process is complete, there are three indivisible blocks left,

$$a^*$$
$$bd_1 l_3 d_3^* c_3^*$$
$$c_1 l_2 d_2^* c_2^* ,$$

as shown in Figure 6.11. These blocks are ordered by decreasing utility to give the optimal strategy,

$$a^* bd_1 l_3 d_3^* c_3^* c_1 l_2 d_2^* c_2^*.$$

6.5 Discussion

6.5.1 Review

In this chapter we have shown how an averaging assumption can be used to provide estimates of the local probability for lookup and inference actions in a search space. These numbers, together with the reduction theorems can be used to assemble the cheapest complete strategy for backward inference problems requiring only a single answer. Although this technique requires searching the inference space, the search does not require examination of conjunctive subspaces for every answer generated. Also, one calculation for a space would suffice for all problems of a given form. In fact, calculations of the maximal utility for each goal form could be done at compile-time as facts are entered into a system. Essentially, this amounts to precomputing and storing the promise of every possible goal form. If this is done, the control overhead during problem solving is only a small constant factor.

6.5.2 Related Work

The idea of using elementary decision theory in the selection of strategies appears to have its origin in the works of Simon and Kadane [SK75] and Sproull [Spr77]. As mentioned in Section 6.4.1, Simon and Kadane proved Theorem 6.6 for the case where local probability is always one. They suggested the use of this result for control of general search problems, but gave no indication of how to estimate the global probability for individual steps. More recently, Barnett [Bar84] has investigated the efficacy of this sort of analysis on some simple examples and has considered the sensitivity of the analysis to errors in the estimation of expected cost and global probability. In a similar vein, Rosenschein and Singh [RS83] have augmented the simple expected cost formulae of Simon and Kadane to include the cost of control reasoning for analysis of this sort.

Sproull [Spr77] makes use of decision theory in planning. He assumes a priori information about the local probability of success for the available actions and uses this information to determine probability of success and expected cost for different possible plans.

There are some important differences between the work described here and the work mentioned above.

1. The analysis is limited to inference actions.

2. We do not assume a priori knowledge of the local or global probability for the inference steps.

In a sense, it is the limitation to inference actions that makes possible the calculation of probabilities for the individual steps.

The problem of strategy selection is not unlike many scheduling problems described in the operations research literature (e.g. [Gar73]). Recently Treitel [Tre86] has begun to investigate the use of linear programming for choosing between forward and backward inference steps. Other techniques and results from operations research may well apply to the problems of optimal strategy selection.

6.5.3 Extending the Results

On the surface it would seem that the methods of computing the promise of an inference step should extend easily to problems involving mixed forward and backward inference, and problems where more than one answer is required. However, there are some significant difficulties involved in handling these more general cases.

Multiple Answers

For the analysis in this chapter, we have only considered problems where a single answer is required. There is a serious difficulty with extending this analysis to multiple answers. Consider calculating the expected cost for a complete strategy when the problem is to find n answers. For each substrategy we must keep track of the probability that the substrategy will produce one answer, two answers, three answers, all the way up to n answers. We must also keep track of the expected cost of a substrategy for any number of answers between one and n. In other words, we are no longer working with single numbers, but with probability and expected cost distributions. While it is still possible to do these calculations, the algebra becomes hideous. It is also not clear how to directly apply the reduction theorems to these distributions. If the utility distribution for one strategy is uniformly less than the utility distribution for another, the theorems would seem to apply. But if the utility distributions cross, it would be necessary to actually carry along a strategy distribution. As an example, consider a search space containing three branches a, b and c, as shown in Figure 6.12, and suppose the problem involves finding two answers. Suppose that branch a is cheapest and may or may not produce an answer. Branch b is more expensive but is guaranteed to produce an answer and branch c is very expensive but is guaranteed to produce two or more answers. Because of its low cost, a is the optimal first step. If an answer is found, b is the best next step. However, if and answer is not found, c must be done anyway in order to solve the problem, so c is the best next step.

There does not appear to be any easy way around this problem. We could assume that the probability distribution was uniform about the expected number of solutions for a strategy and choose a single strategy accordingly. However,

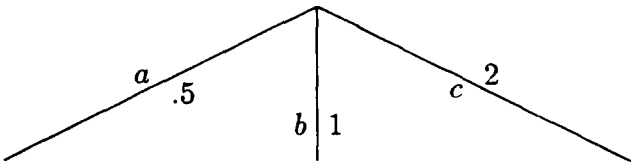


Figure 6.12: Multiple answer example

it is not clear that this additional averaging assumption is a good one. A functional goal expression would have a very different shaped distribution than one like $Child(Martha, x)$. In such cases any single strategy might be suboptimal as was the case in the abstract example above.

6.5.4 The Independence Assumption

Throughout this chapter we have assumed that inference actions in separate branches of the search space have independent cost and probability of success. In other words, the performance of one such inference action does not affect the probability of success or the expected cost of any other inference action that is not a descendent of it. There are at least two cases where inference actions can violate the independence assumption.

First of all, if a portion B of the search space is known to be redundant with another portion A, B will not be independent of A. Suppose that a is a complete strategy for the portion A and b is a complete strategy for the portion B. For the strategy ab the likelihood of success for the second step b is reduced to zero since B is redundant with A. However, for the strategy ba the likelihood of success for both strategies is unaffected by the redundancy. The matter is more complicated for strategies that interleave searching of the two portions. In this case, any actions from B that follow all actions from A will have their likelihood reduced to zero.

While it is possible to take account of known redundancy in calculating the expected cost and probability of success for strategies, the reduction theorems of Section 6.4 may no longer hold. It is not clear whether versions of these theorems can be developed that will still lead to the optimal strategy when redundancy is present. There is at least one special case result that holds under these circumstances. If the steps in the portion of the space that subsumes the redundant portion form an indivisible block and the maximal utility for the redundant portion has lower utility than this indivisible block, the reduction will be unaffected and the redundant portion can safely be deleted. Even when this is not the case, the recognition of redundancy can only improve a strategy. Thus, if the optimal strategy is formed by

ignoring the redundant portion, and appropriate pieces of the redundant portions are then deleted from the strategy, the strategy, although perhaps not optimal, will be at least as good as the optimal strategy if the redundancy had not been recognized.

The second situation where the independence assumption is violated is when caching of intermediate solutions is used, and similar subgoals appear in different portions of the inference space. If answers to a subgoal g are cached and another similar subgoal g' appears elsewhere in the search space, the overall cost of finding answers to g' has been reduced. More precisely, the probability of finding answers to g' in the database has been improved, and some of the inference steps for finding answers to g' become redundant. It is relatively easy to construct examples where the addition of caching can drastically change the optimal strategy. Again, calculating expected cost and probability of success is not the problem here. The problem is that the reduction theorems no longer hold under these circumstances. It is not clear whether more powerful reduction theorems can be found that will cover this situation. However caching can only improve a strategy, so the strategy found by the reduction theorems may be good enough for most cases.

6.5.5 Conjunctions

In the kinship example used in this chapter, we actually only considered one of the many possible orderings for the conjunctive clauses. The reason is that all of the different possible strategies for enumerating the solutions to a conjunction are redundant. As with other redundancy, the optimal way of finding answers to a conjunction might involve getting some of the answers by one ordering, and others by a different ordering.[5] In order to use the reduction theorems as is, we would have to make the assumption that a single (static) ordering would be optimal for conjunctions. As with the general problem of redundancy, there may be no simple extension to the reduction theorems that will solve this problem. If the static ordering assumption is not reasonable a much more expensive search of the strategy space will probably be necessary.

6.5.6 Graphs and Forward Inference

There is no fundamental reason why similar calculations could not be made for expected cost and probability of success of forward inference or for strategies containing both forward and backward inference steps. For a forward inference step, instead of computing the probability that a goal expression would match a fact in the database, or the right hand side of a rule, we would compute the probability that

[5]Moore [Moo75] has found examples where this is, in fact, the case.

a fact would match the a premise of a particular rule or one of the goal expressions. The most serious difficulty is that when forward inference steps are considered, the inference space becomes a graph instead of just a tree. This means that Garey's reductions will not completely reduce the space and some search will be necessary in order to find the optimal strategy.

However, it is still possible to make use of Garey's results for graphs, although some caution must be exercised. Again, starting at leaf nodes, we work up through the space computing maximal best strategies. For those portions that are trees, Garey's results apply, allowing rapid computation of the maximal best strategies. Where the space is not a tree, search will be required to discover the maximal best strategies. However, once a maximal best strategy has been formed, the reduced space may again be tree-like, allowing the use of the more powerful reduction theorems. In short, construction of the optimal strategy proceeds from leaf nodes upward taking advantage of the powerful reduction theorems wherever possible, and falling back on limited search wherever a portion of the space becomes a graph.

Treitel [Tre86] has been investigating the use of linear programming for finding optimal strategies to problems. It is possible that a similar approach would prove advantageous here.

Chapter 7

The Efficacy of Control

In the previous chapters we developed a theory of how semi-independent control could be provided for an inference task, and illustrated this with several detailed examples. Although we gave arguments that semi-independent control was necessary in each of the examples, the cost of this control, or the most appropriate time to make control decisions was not considered.

7.1 The Run-time/Compile-time Spectrum

There is a spectrum of possible times when control decisions can be made, from the time that a system is constructed, until the time immediately prior to when a control decision takes effect. We refer to these two extremes as *compile-time* and *run-time* control respectively.

There are several trade-offs involved in deciding when control decisions should be made. The later a control decision takes place, the more specific and accurate it can be. Compile-time control decisions require predictions of the circumstances and frequencies with which different problems will occur. In contrast, run-time control decisions do not require any estimation of this sort. On the other hand, run-time control involves greater overhead because it multiplies the complexity of the control decision process by the complexity of the inference process.

As an illustration, consider again the case of ordering conjunctions (Chapter 5). Given a fact with a conjunctive premise, there are three interesting times when control could be provided. The first possibility is to provide control at compile-time when a rule is first entered into the system. This involves predicting how the rule will be used and determining the best ordering of the premise clauses for each case. For this approach, the ordering process is only done once. The overhead is therefore amortized over the life of the system. On the other hand, determining the best ordering(s) requires estimating how the rule will be used and what facts will

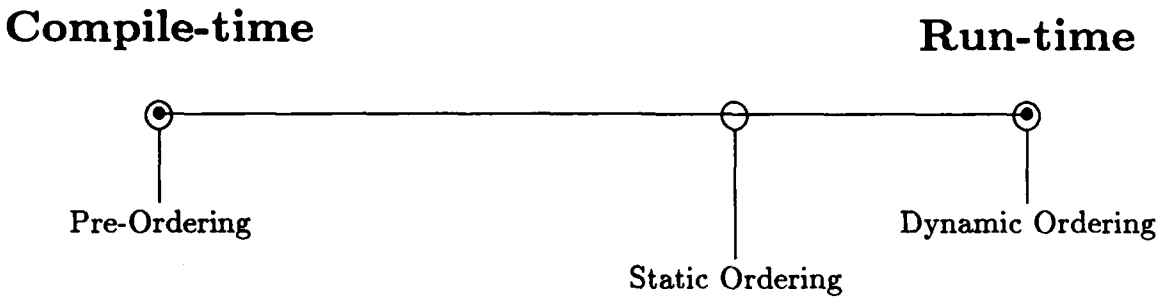


Figure 7.1: The Run-time/Compile-time Spectrum.

be available in the database when the conjunctive premise is encountered. These estimates are difficult to make and may not be accurate. As a result, such control decisions may not be sufficient to give acceptable run-time performance.

The second possibility is to order the conjunctive premise each time it is encountered in the problem solving process, as we did for the intelligent agent's problem of Chapter 5. The overhead will be much higher if this conjunctive premise is encountered frequently. On the other hand, the control is much more likely to be effective, because the exact characteristics of the problem and the database are known at the time the conjunctive premise is encountered. The only estimation required is in computing the number of solutions to conjuncts knowing the variables that will be bound, but not the actual bindings.

The third possibility, extreme run-time control, is to decide conjunct order during the actual generate and test process. After the solutions to one conjunct are found, the best remaining conjunct is determined for each set of variable bindings to the conjuncts already solved.[1] This is an expensive process because a control decision must be made for each node in the conjunctive search space. On the other hand, such control is the most accurate and can lead to a considerably smaller search space than either compile-time control or the intermediate approach.

The three different possibilities are illustrated in the spectrum shown in Figure 7.1. These three possibilities are the most interesting points on the spectrum because they correspond to the times when new information relevant to the control decision becomes available. In general, the interesting times are those when maximum information is available without a corresponding increase in control overhead.

The general lesson from this example is that the greater the control accuracy, the higher the overhead. In evaluating this trade-off there are several criteria that must be considered,

- the extent to which optimal control is needed for the problems the system is

[1] *Dynamic* conjunct ordering was discussed in Section 5.4.2

expected to solve,

- the effectiveness of control decisions made at each possible time, and
- the overhead involved in control decisions at each possible time.

The impact of these three factors is clear. If very accurate control is unnecessary, compile-time control is often adequate. Likewise, if the overhead of run-time control is high, compile-time control is preferred. However, if compile-time control is not good enough for even a small subset of the expected problems, more expensive run-time control is necessary. The diversity of problems expected by a system, and the frequency with which different problems occur have a major impact on this latter factor. If the majority of problems expected by a system fall into a few simple categories, compile-time control is more likely to be effective. However, if the space of possible problems is infinite and problems do not occur with any regularity, compile-time control has little chance of success.

Of course, run-time and compile-time control are not mutually exclusive. If there is some subset of problems that occur frequently, and compile-time control is effective for that set of problems, compile-time control should be used in conjunction with run-time control. For the case of the intelligent agent it makes sense to compute and save the best ordering for common conjunctive queries at compile-time. Run-time control would remain as a safety net for less frequent problems not covered by compile-time control decisions.

7.2 The Cost of Control

For inference engines with varying degrees of control, the expected cost of solving a problem can be plotted against the depth of reasoning required to solve the problem. For an inference engine without control the expected cost of solving a problem rises exponentially with problem difficulty. The rate of rise depends upon the average breadth of the search space. This is illustrated by the steepest curve of Figure 7.2. This curve represents only an expected value. The cost of solving any particular problem could be a linear function of the depth of search required, or it could be infinite, depending on how quickly the inference engine stumbles onto an answer. But, on the average, the cost will be exponential with the depth of search required.

In contrast, if an inference engine is perfectly controlled (no choice involved at each level of the search) the expected cost is linear in problem depth as illustrated by the lowest curve in Figure 7.2. For less perfect control the inference cost is still exponential, but less dramatically so than for uncontrolled inference. The cost curve lies somewhere between the curves for perfect control and no control. The steepness of the exponential is an indication of effectiveness of the control. Compile-time

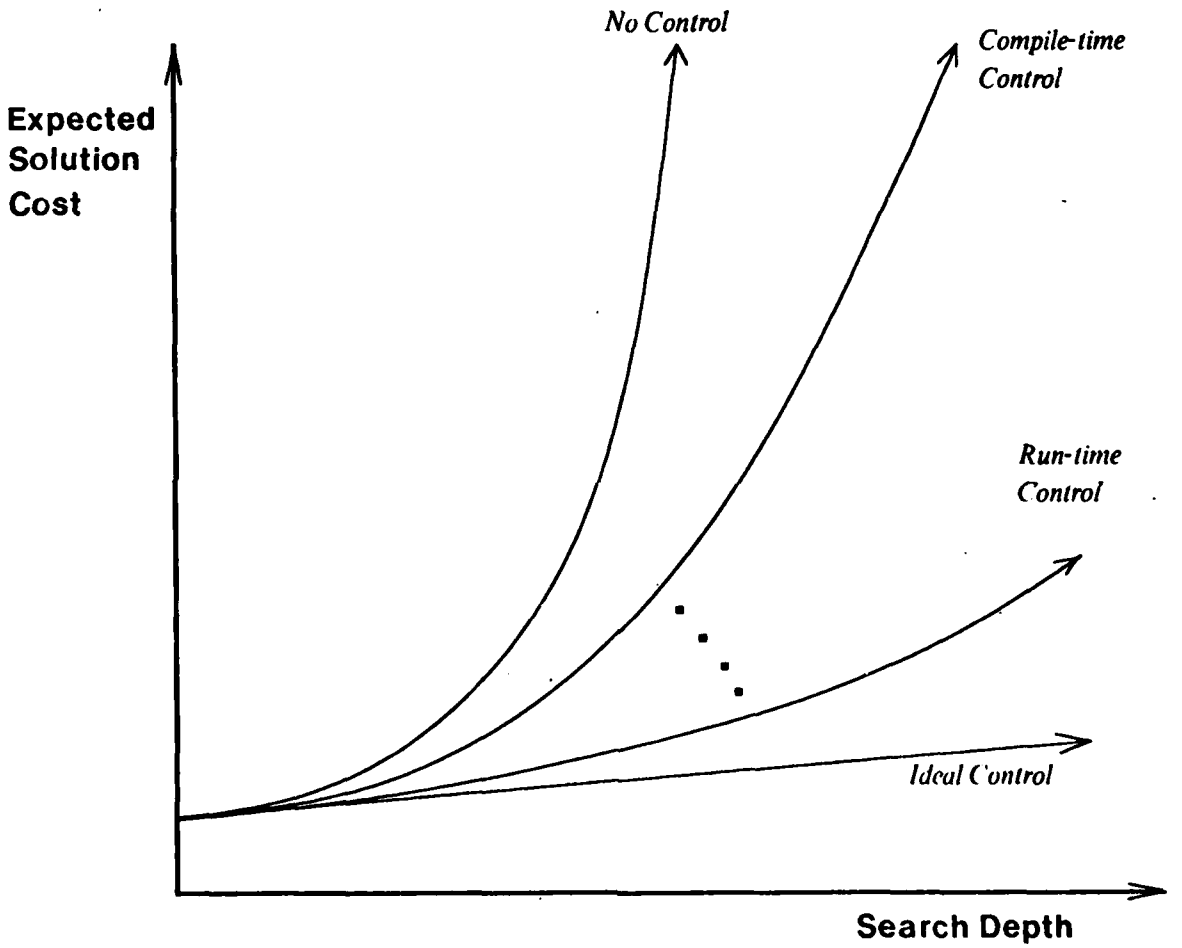


Figure 7.2: Cost as a function of problem difficulty for controlled and uncontrolled inference engines.

control is generally less accurate and therefore less effective than run-time control, so its exponential has a higher slope than that for run-time control.

Control of inference is not without cost. In the curves of Figure 7.2, the cost of providing control to the inference engine was completely omitted. Perfect control requires comparing the cost and effectiveness of all possible strategies for each individual problem. This requires more effort than a complete search of the inference space. As a result, the combined problem solving and control cost for an inference engine with perfect control is an even steeper exponential than that for an inference engine with no control. This is illustrated by the steep dashed exponential of Figure 7.3. For compile-time control, the control decisions may require considerable computation, but this cost is amortized over the entire life of the system. Compile-time control therefore adds a constant factor to the cost of solving every problem. The combined cost curve for compile-time control is illustrated by the middle dashed curve in Figure 7.3. For run-time control, like that presented in the case studies, control cost is exponential in problem difficulty. However, unlike perfect control, the slope of the exponential is much lower than the slope of the exponential for uncontrolled inference. For example, consider the case of ordering conjunctions. The cost of ordering is exponential in the number of conjuncts. Likewise, the cost of finding solutions to a conjunction is exponential in the number of conjuncts. But since there are many solutions to each conjunct, the latter exponential is very much larger than the exponential cost of ordering. The combination of inference cost and control cost is therefore the sum of two exponentials, as illustrated by the remaining dashed curve of Figure 7.3.

As the curves of Figure 7.3 show, for simple problems we are better off with uncontrolled inference than with sophisticated control. In such cases, sophisticated control is generally not cost-effective. In fact, the large overhead of control may be completely unacceptable.

This is not the case for more difficult problems. There is a break-even point where control begins to become profitable. As problem difficulty increases beyond the break-even point, the likelihood that control will be needed is greater, and the expected payoff from providing control is greater. For truly difficult problems, control is absolutely essential. Without it, inference can take an unacceptably long time. Thus, adequate control must be provided if an intelligent system is expected to handle difficult problems, regardless of how infrequently they are encountered. This is the fundamental dilemma; powerful control is essential for difficult problems, yet is sufficiently costly that it is often impractical for simpler problems. For the case of recursive inference this dilemma becomes quite apparent. Control involves difficult proofs that portions of a search space do not contain novel answers. It is completely impractical to attempt such proofs for every subgoal generated. However, such control is absolutely necessary. For a recursively infinite search space, failure to

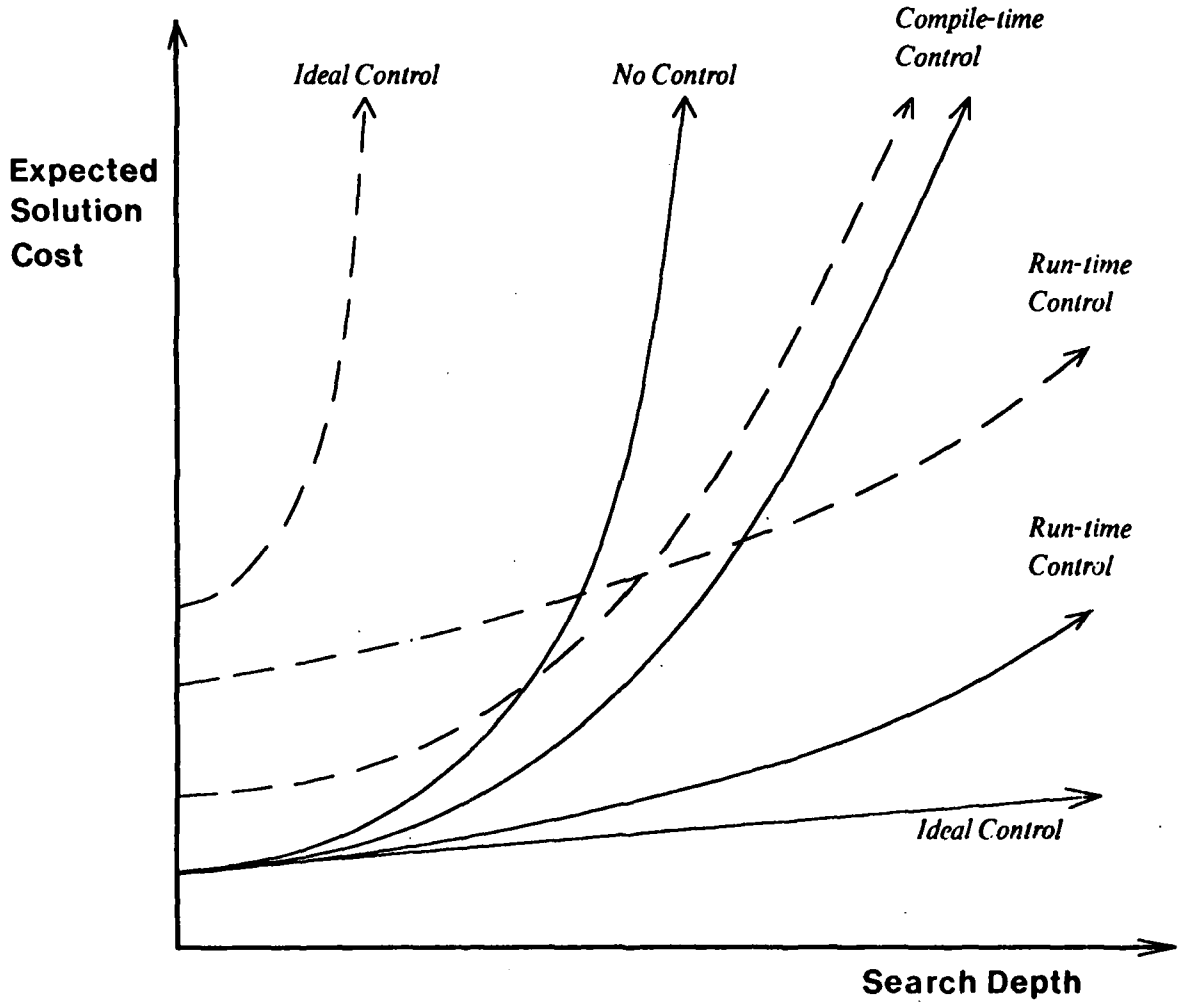


Figure 7.3: Cost as a function of problem difficulty for controlled and uncontrolled inference engines.

control inference can be disastrous.

7.3 Controlling Control

The solution to the control dilemma is to limit the application of costly control methods – to "control" control. One might consider this to be *meta-control.* However, this is not the usual sense of meta-control. Here we are referring to control over the invocation and duration of control, not to the control of explicit inference involved in control.

There are three criteria that must be satisfied by any process of controlling or mediating control:

1. The mediation itself must be inexpensive. In other words the overhead of mediation must not be unacceptable for simple problems.

2. The mediation must not permit the overhead of control to become unacceptable for simple problems.

3. The mediation must be *conservative.* It must institute adequate control for all problems that require control (i.e.: would be unacceptably costly to solve without control).

Interestingly enough, there are many different ways of mediating control that satisfy these criteria.

7.3.1 Interleaving

The simplest mediation procedures are those that involve a predetermined interleaving of problem solving and control. For example, consider the process of devoting one unit of time to problem solving, then one unit to control, then another unit to problem solving, and so forth. If the cycles devoted to control produce any information about what strategy to pursue, the information is passed on to the problem solver, which will take advantage of this information on subsequent cycles. If the problem solving succeeds, the process terminates.

Although interleaving may seem a bit schizophrenic, it has several nice properties. First of all, it is inexpensive, provided the cycle-times are not too short. Secondly, it is conservative, since it guarantees that control will be provided for difficult problems. Finally, it guarantees that the solution time will not be more than twice the necessary amount. For example, if the problem does not benefit from control and takes n problem solving cycles the total time spent will be n problem solving cycles plus $n - 1$ control cycles for a total cost of $2n - 1$. Alternatively,

suppose that control is cost-effective and n control cycles are required to find the right strategy, which will take m problem solving cycles. Then, n problem solving cycles may be wasted initially until the control result is obtained. This will give a total cost of $2n + m$. The best possible is $n + m$, so the result is still no more than twice optimal.

Many variations on this mediation technique are possible. First of all the times allocated to problem solving and control in a cycle need not be identical. Ten cycles could be spent on problem solving for every one spent on control. Of course, this changes the characteristics of the process. For simple problems that do not benefit from control, the cost will be only ten percent higher than optimal. However, for difficult problems, the cost may be as much as ten times the optimal cost. If few difficult problems are expected by a system, a lopsided allocation like this one, may be entirely appropriate.

Another variation is that not all cycles must have the same proportions of problem solving time to control time. For example, consider the procedure that spends 9 units on problem solving, followed by 1 unit on control, followed by 8 units on problem solving, followed by 2 units on control and so on, down to the point where equal time is spent on problem solving and control (or perhaps a disproportionate amount on control). It is more difficult to estimate the effect of such a procedure. For very simple problems that do not benefit from control, the cost will not exceed ten percent of the optimal cost. For difficult problems that require control, cost will be no worse than twice optimal, plus a constant amount $(8+7+6+5+4+3+2+1)$ for the problem solving time wasted initially. Such a strategy is appealing, since the constant factor is likely to be insignificant for difficult problems.

Thresholds

An extreme version of interleaving is something that might be called *control thresholding*. In control thresholding, problem solving is done for some specified period, then control is instituted and allowed to proceed unimpeded. Thresholding is an intuitively appealing mediation procedure. It can be thought of as a frustration mechanism for a problem solver. When the frustration becomes high enough, the problem solver jumps to the control level and considers how to solve the problem. In the cost diagram, Figure 7.3, this corresponds to following the curve for an uncontrolled inference engine until the cost threshold is reached, and then jumping to the curve for controlled inference.[2] The closer the cost threshold is to the break-even point, the better the problem solver will perform. If the threshold is set too low the discontinuity will occur to the left of the break-even point and control is invoked

[2] Actually, this jump is slightly higher, since effort has already been expended on problem solving and some of this effort may be wasted.

for problems where it is not cost-effective, slowing down the inference process. Alternatively, if the threshold is set too high, the discontinuity will be to the right of the break-even point and problems that could benefit from control will not receive it, thus wasting problem solving energy. It is therefore important to choose the detection threshold and the method of measuring cost carefully, based on analysis of the expected cost-benefit of the control procedure and the frequency with which difficult problems occur.

A control thresholding process need not be limited to a single threshold. For example, a simple control mechanism could be implemented after a first threshold is reached and a more costly, but more accurate, control mechanism could be implemented when a second threshold is reached. In Chapters 1 and 2 we argued that weak domain-independent control methods are inadequate. In fact, such methods may very well have a place in problem solving when backed by more powerful control. For example, in the case of conjunct ordering, an appropriate strategy might be to perform blind generate and test for some number of cycles, then use the cheapest-first heuristic to provide a crude ordering of the conjuncts. If, after ordering, the cost of generate and test exceeds another threshold, a search of the space of possible orderings could be instituted.

Cost Monitoring

All interleaving strategies rely on the ability to monitor problem solving and control cost at run-time. As a result, they must measure elapsed time or cost in some fashion. There are many ways to do this.

The simplest is to keep track of elapsed CPU time or real time. Such facilities are usually readily available in most computing environments. A similar method is to keep track of the number of primitive inference steps used in problem solving and control. All of these possibilities are ways of measuring the overall size of the problem solving search space and the control search space. A slightly more elaborate possibility is to keep track of the total number of leaf nodes in the search space, and use the increase in the total number of leaf nodes as the interleaving criterion. This has the interesting property that problem solving will continue as long as the number of alternatives remains small. Likewise, control will continue as long as the number of strategy alternatives remains small. This approach is worth investigating since it may provide a better measure of the likelihood that control will be cost-effective for a problem.

The Pros and Cons of Interleaving

There are both positive and negative aspects to all forms of interleaving. On the negative side, since the institution of control comes only after problem solving effort

has been expended, the advantage of control is not fully exploited. Search that is already done might be unnecessary if control were instituted in the beginning. Furthermore, elapsed time is a measure of how much work has already been done on a problem, not how much work remains to be done. As a result, control may be invoked in cases where very little remains to be done. On the positive side, interleaving is inexpensive, simple to implement and is applicable to any control procedure.

7.3.2 Assessing Problem Difficulty

One might regard interleaving methods as rather crude syntactic hacks. After all, they only discover that control is needed after time has already been wasted on trying to solve a problem. Ideally, we would like to quickly evaluate each problem before problem solving begins, to determine whether or not control will be cost-effective. Control could then be instituted only when it would be cost-effective. This is unrealistic. An accurate assessment of whether or not control is cost-effective would usually be very expensive, violating the first of the three criteria for mediation procedures.

However, there are some circumstances where such assessments are not overly expensive. If there is a body of domain knowledge available about which problems are easy or hard, this knowledge can be exploited. For example, in the operating systems domain we might give a problem solver the information that all conjunctions containing the *File* predicate are potentially hard. In such cases, control could be provided immediately.

Sometimes more general principles can also be used to recognize simple or difficult problems. In the case of recursive inference no recursion is possible for facts that are not a part of a recursive collection. If such collections are recognized and marked at compile-time this information can be used to avoid control for simple problems. Likewise, in controlling backward inference, if we were to calculate the maximum possible inference depth for every relation (at compile-time), this information could be used to quickly recognize potentially difficult problems.

When such assessments are used, the mediation procedure must still be conservative. If there is a large body of information about which problems are easy, a reasonable approach is to weed out the easy problems and institute control for everything else. If the information is less complete, it becomes necessary to use assessment in conjunction with some form of interleaving.

There are also many possible hybrids of assessment and interleaving that deserve study. For example, inference could be allowed to proceed for some number of cycles, followed by an assessment of the cost-effectiveness of control, followed by some deliberate strategy of mixing inference and control, perhaps followed by more

assessment, and so on. The possibilities are unlimited here, but we enter the realm of speculation and fantasy. More experience with the simple possibilities must be gained before such complicated alternatives can reasonably be considered and studied.

Chapter 8

Discussion

8.1 Implementation Issues

8.1.1 Design of the Inference Engine

In this dissertation we have concentrated on the knowledge and reasoning involved in making good control decisions for particular problem solving situations. This matter was considered as if the inference engine were capable of following any strategy recommended by the control process. In designing an inference engine, one must take into account and allow the range of different possible strategies that can be recommended by any control processes used. The range of control strategies expected therefore has implications for the structure of an inference procedure.

The control methods developed in the first three case studies impose only minor restrictions on the design of an inference engine. For example, the ability to take advice about conjunct order demands only that the inference engine have an explicit representation of conjunctions and that it be able to process conjuncts in the order indicated by the control procedure. Conjunct ordering can therefore be added to virtually any inference engine with minimal modification. However, the matter is not so simple for the kinds of strategic advice considered in the final case study (Chapter 6). To follow such advice, an inference procedure must be capable of performing backward inference steps in an arbitrary order. This means that the inference procedure must be able to work on any of the active subgoals at any given time, regardless of their location in the search space. The set of subgoals must therefore be kept in a data structure that is completely accessable, like an agenda, or a spaghetti stack. Most depth-first backward chainers could not make use of such control information because the set of subgoals is stored implicitly in the program stack (which usually cannot be accessed or modified in arbitrary ways).

Interleaving inference and control requires additional flexibility on the part of

the inference engine. It requires the ability to suspend and resume the inference process, and also requires the ability to follow a new strategy when inference is resumed. As an example, consider the case of conjunct ordering for the conjunction $A \wedge B \wedge C \wedge D$. Suppose that the inference engine begins by enumerating the solutions to A, B and then C. Then suppose that the inference is interrupted due to the size of the search space, and control reasoning leads to the strategy recommendation that the conjuncts should be solved in the order $\langle A, C, D, B \rangle$. When inference is resumed the generate and test engine must be capable of using the solutions already found to A, but process the remaining conjuncts B, C and D in the recommended order. Thus, the generate and test engine must be able to explore any of the different possible ways of enumerating solutions to a conjunction at any (interruptible) point during the generate and test process.[1]

More generally, when inference is resumed after control reasoning, the preference ordering on possible inference steps has been changed or augmented. To take advantage of this control information the inference engine must be able to perform the new "best" inference step indicated by the partial ordering.

The general issue here, is the ability of an inference engine to follow arbitrary advice about strategy. The extreme is an inference engine that can perform any legal forward or backward inference step at any time. We say that such a procedure is *completely cooperative*. The construction of such a procedure (first advocated by McCarthy in [McC68]) is discussed in [GS85]. While such an inference engine may be important ultimately, none of the cases considered here require quite this much flexibility.

8.1.2 The Control Procedure

In the case studies, control knowledge was expressed as explicit meta-level axioms. The reason for doing so was primarily pedagogical. Explicit representation of control knowledge makes clear the essential relations and concepts involved, leading to an understanding of the logical steps involved in particular control decisions.

However, there is no essential reason why the control principles developed here must be implemented as explicit meta-level reasoning. We could instead choose to implement them as procedures in languages like LISP or PROLOG. There are advantages and disadvantages to both alternatives.

Many authors have argued that control should be implemented as explicit meta-level reasoning using a database of control knowledge.[2] There are two basic arguments for explicit meta-level reasoning. First, the explicit representation of knowledge involved in control decisions facilitates the implementation and modification

[1] This requires that the state of the generate and test engine be recorded explicitly.

[2] For example, see [McC68], [Hay73], [McD77], [dDSS79], [Dav80], [CL81], [Doy80] or [GS82].

of control. Second, explicit representation facilitates the explanation of control decisions. These reasons are, in fact, exactly the same reasons used to support the position that ordinary base-level procedures should be implemented as a reasoning process using an explicit declarative data base.

The disadvantage of the explicit approach is in the efficiency of the control process. If control is implemented as explicit meta-level reasoning we must concern ourselves with controlling the meta-level inference process. This is not a matter to be taken lightly. For example, consider the axioms for a simple control strategy like the cheapest-first strategy for conjunct ordering. Reasoning with axioms of this complexity requires considerable control guidance. Of course, the control principles developed here also apply to meta-level inference. This pushes the problem up another level, but doesn't solve it. Alternatively, we could provide explicit control information about how to use the control axioms for different control problems. One can think of this as domain-specific control information at the meta-meta-level, except the domain is that of control problems. Still, some control may be required at the third level in order to efficiently reach control decisions, even with these much simpler axioms.

A reasonable compromise to the trade-off between explicitness and overhead is to build a certain amount of control into the inference engine used at some level. A good candidate for this is Lock resolution [Boy71,Gen85a]. Lock resolution still permits general forward and backward inference but allows control of the inference process by the order in which clauses and rules are specified.[3] In this way Lock resolution allows the specification of control information necessary for efficient processing of the meta-level control axioms, while preserving the declarative character of the control axioms.

8.2 Contributions

The most tangible contribution of this research is the collection of control principles developed in the four case studies. For example, in Chapter 4, the various theorems and corollaries for controlling general and special cases of repeating inference, and Theorem 4.16 for controlling divergent inference are powerful and novel results. In Chapter 5, the adjacency restriction (Theorem 5.2) and its corollaries are powerful results for the ordering of conjunctive queries. In Chapter 6, the utility analysis of backward inference paths and the application of Garey's theorem (6.7) and Simon and Kadane's theorem (6.6) to the selection of optimal inference strategies appears to be new.

A somewhat less tangible contribution (but in my view an equally important one)

[3] PROLOG permits this for backward inference, but not forward inference.

is the general approach for solving control problems. We have viewed control as reduction and search of the space of possible strategies. To evaluate different strategies we make an averaging assumption and use basic utility theory to compute expected cost and probability of success for the strategies. This computation ultimately depends upon having information about characteristics of the system's database; the domains of relations, sizes of sets, and other properties such as the transitivity and monotonicity of relations. This approach to control is in contrast with the use of either domain-specific control information or weak domain-independent control information.

8.3 Related Issues

8.3.1 The Impact of Advanced Hardware

Current efforts to produce super-computer and massively parallel machines may have some eventual impact on this work. For example, in a machine with even two processors it would be possible to devote separate processors to base-level and meta-level inference. This would make elaborate interleaving schemes, like those discussed in Chapter 7, unnecessary.

Alternatively, if computational speed were to increase by two orders of magnitude the control and problem solving costs for easy problems might become computationally insignificant. In this event, the added overhead of sophisticated control may become a negligible burden for simple problems. This, too, would make the elaborate interleaving schemes of Chapter 7 unnecessary. Control could simply be applied universally to all problems. Thus, the discussion of interleaving strategies is potentially vulnerable to improvements in hardware.

However, the remainder of the dissertation is more immune to such technological development. Massively parallel hardware or super-computers will never solve the control problem. Combinatorial explosion can not be beaten by finite improvement in computational speed or by any fixed number of processors. In some sense, the advent of parallel machines makes the control problem harder, because one must also consider how to distribute inference over the processors.

8.3.2 Automatic Control and Common Sense

The term "common sense" is often used to refer to the sort of reasoning behavior that a typical person might display while driving to the grocery store. As Webster's Dictionary puts it, *something that is evident by the natural light of reason and hence common to all men.* By this definition, common sense is a conglomerate of many different facts and mechanisms. A crucial component of what we call common sense

is a collection of knowledge about the stuff of everyday life; such things as waffle irons, baseballs, used car salesmen and the effects of every day actions. Equally important, is a large collection of knowledge about how to solve problems; in other words, control knowledge. For example, if we asked a colleague to find the files requested in the intelligent agent's problem (page 90) and two days later he is still methodically searching through a complete file listing, examining every file, we would accuse him of being a dunce and of having no common sense whatsoever. It isn't that our dunce doesn't know enough about operating systems. Rather our dunce doesn' do any analysis to decide what is the best approach to solving the problem. Analysis of utility is the key to smart problem solving, and smart problem solving is one of the key features we attribute to common sense.

8.3.3 Psychological Relevance

Intuition and introspection have been used liberally as a source of inspiration for developing the control theories in this research. However no scientific statements about psychological validity are intended or implied. As with most AI research, it would be interesting to know the extent to which these control strategies agree with the strategies employed by humans. However, the computational characteristics of people and current machines are still quite different and it would be wrong to assume that the best strategy for one would necessarily be the best strategy for the other. As discussed in Section 7, the decision about which control strategy to employ at what time depends heavily on the costs involved in looking up items from the database and the costs of performing various inference steps. There is no question that these costs are different for people and machines, or that they differ for different people. It would, therefore, not be surprising to find that humans do not apply the same strategies for the same problems as we have suggested for a machine.

8.4 Methodology

Although the analysis and results in this dissertation are theoretical, the research methodology has been more of an empirical one. The case studies in this dissertation resulted from the analysis of specific reasoning traces. The traces came from several different projects at Stanford aimed at the construction of intelligent systems for performing specific tasks. The control problems that resulted in the first case study (Chapter 3) arose simultaneously in two different domains, a system for modelling renal physiology constructed by John Kunz [Kun84] and a system for income tax consultation (intended for pedagogical purposes in a laboratory class at Stanford) built by Mike Genesereth [Gen83]. The control problems that gave rise to the case

study on recursive inference arose in building systems for reasoning about the behavior of circuits from models of their structure [Gen85b,Sin85,Kra84]. In these systems there is often a need to express and reason with definitional statements like the relationship between the input and output characteristics of a device. There was also a need to express symmetry and transitivity relationships, all of which resulted in forms of recursive inference. The difficult problem of controlling divergent inference arose in two projects for reasoning about loops in sequential circuits [Sin85,Kra84]. Control problems with conjunct ordering have arisen in several applications here at Stanford, but the problem became paramount in the Intelligent Agent project at Stanford [FGKM80]. Here one has no control over the order in which conjunctions are specified, so there are no alternatives to automatic control. The problem also arose later in another project for reasoning about the behavior of operating systems [Die84]. Finally, the case study on backward inference arose from a system for generating tests for faulty combinational circuits [Sin85].

In all cases we have started with specific control problems from real systems. We then analyzed the reasoning for the problematic example(s) to determine what the ideal control decisions would be. In particular we attempted to identify those portions of the reasoning that a person would regard as silly. The next step was to ask *why* we regarded this reasoning behavior as silly. In other words we wanted to find and precisely characterize the rationale behind the appropriate control decision. Our initial theory was sometimes wrong. For example, in the case of conjunct ordering my initial hypothesis was that the cheapest-first heuristic was a correct theorem. At this stage the hypothesis must be subjected to extreme examples to determine the circumstance in which it applies. As in the case of conjunct ordering this often leads to a better hypothesis that contains the original as a special case.

In fact this dissertation represents only the first portion of the research methodology. The final step, which is yet to be performed, is the implementation and testing of these results. The reason for this absence of implementation is that, until recently, it has not been clear how to efficiently limit the application of control to those cases where it was likely to be cost-effective. Interleaving problem solving and control appears to be the solution to this difficulty. Once it is implemented, full implementation and testing of the strategies for the case studies will become possible.

Note that this methodology is essentially an empirical one. It begins with problems that arise in the construction of systems and ends with implementation and testing of the results. A reasonable phrase for this methodology is *meta-knowledge engineering*. There is an aquisition phase, where knowledge about control is elicited from an "expert" at problem solving, followed by precise characterization of this knowledge, followed by implementation and testing.

8.5 Some Open Questions

8.5.1 Empirical Questions

There are many questions raised in this research that can only be answered by studying an implementation. For example, the efficiency and effectiveness of the various control principles developed here needs to be verified by studying performance on a large and diverse mix of inference problems. Likewise, the various interleaving strategies proposed in Chapter 7 require implementation and empirical study to determine their performance for different problem mixtures. Data and experience from such implementation would also provide a basis for studying the problem of how to automatically divide time between control and inference. At present, there is no data or experience with the control/inference trade-off and, as a result, we can do little more than speculate about the laws of allocating time between inference and control.

Another interesting question that cannot be answered yet (and may not be answerable for many years yet) is the frequency of problems that can be handled with compile-time control. Although we have concentrated on run-time control here, some of the analysis in the case studies can also be applied to compile-time control. Treitel [Tre86] is currently investigating many of these possibilities. It will be interesting to see what fraction of inference problems are handled effectively using compile-time control. Equally interesting is the extent to which run-time control using averaging assumptions will be effective for the remainder. In both the case studies for conjunct ordering (Chapter 5) and backward inference (Chapter 6), we used averaging assumptions in estimating the utility of different strategies. These assumptions lead us to a strategy that is best for an "average" problem of the given form. There may well be situations where this average strategy is not good enough. Imagine a situation where the optimal strategy for finding siblings of one set of individuals is quite different from the optimal strategy for finding siblings for the other set. The optimal strategy for an "average" individual may not be a very good strategy for either set. It remains to be seen how frequently such situations arise in practice and how sensitive the average strategy is to individual variation among members of a set.

8.5.2 Theoretical Questions

In addition to these empirical questions, there are several interesting technical questions that remain unsolved.

Higher order statistics

In Section 5.4.2 we discussed the possibility of dynamic ordering for conjunctions when the average strategy is not good enough. Another possibility worth considering is the use of higher order statistics like variance in the determination of strategy. Suppose that, in addition to the average number of solutions to a proposition (for individuals in a given set), we knew the variance in the number of solutions (for members in that set). Using this information, the range of performance expected using a given strategy could be predicted. In cases where the variance was high, and the performance of a single average strategy is insufficient, we could consider dividing the set into pieces and using different strategies for the different pieces. This is in contrast to dynamic ordering where a separate strategy would be chosen for each member of the set. Although such possibilities are interesting to consider it seems premature to devote too much effort to their development until some experience is gained with the actual performance of strategies based on the averaging assumption.

The General Case

The most important technical question is whether the general approach outlined in Chapter 2 and used in the case studies will work for the general case of controlling mixed forward and backward inference involving conjunctions and redundancy. At the end of the case studies on conjunct ordering and backward inference, we discussed the rather significant difficulties involved in extending the analysis to the general case. The mathematics becomes considerably more difficult and the computational complexity of finding optimal strategies continues to increase. It is not clear whether more powerful reduction theorems can be found that will reduce this complexity. If such reductions cannot be found, we may be forced to settle for strategies that are "nearly" optimal. For example, suppose that we do not consider strategies involving the interleaving of inference steps from redundant portions of a search space. For conjunct ordering this would mean considering only strategies where all of the answers are generated using a single ordering. Under this assumption, the independence assumption is preserved, and the analysis of Chapter 6 can be extended to handle conjunct ordering. Other such assumptions may allow the extension of this analysis to multiple answers and mixed forward and backward inference. It is unclear how strategies found using these assumptions will compare in performance to the optimal strategies.

8.5.3 The Science of Control

In the case studies examined here, we have developed a small handful of control principles. The broader scientific question is how many more useful control principles are there? Are there only a few, or are there thousands? In every reasoning trace that we have examined, so far, another control principle has been found. If the analysis for backward inference can be extended to cover the general case of mixed inference, conjunctions, and multiple answers, we would have a complete theory for the searching and reduction of the strategy space. In this event, the case study on conjunct ordering would become only a special case and it would sreem that we were beginning to achieve some degree of closure. However, there may be other powerful special case reduction theorems and results waiting to be discovered.

The other two case studies present somewhat more of a mystery. They involve the recognition of redundancy in the search space. We do not expect to find a more general theorem that subsumes these two, and think it is likely that there are more special case results for the recognition of redundancy.

In a sense, we should be more surprised if there turn out to be only a few such domain-independent control principles. Inference is a difficult task that humans do very well, even when the domain or problem is unfamiliar. From the history of AI we should expect that a large amount of control knowledge must be brought to bear to do it well. This is entirely analogous to the problem of building any expert system, where large amounts of domain knowledge are required to solve non-trivial problems. Only in this case the knowledge is about the domain of performing inference.

Appendix A

Domain-dependence

Although there is a clear intuitive notion of what is meant by the terms *domain-dependent* and *domain-independent*, these notions are difficult to define rigorously. The trouble is that we must separate those facts that are *truly* domain-dependent from those that are tautological and simply contain domain symbols. The definitions therefore take on a model-theoretic character.

First we define what it means for a base-level fact to be domain-dependent. Let I be the intended interpretation of the set of symbols S in a theory T. Let S_D refer to the subset of symbols in S that belong to domain D (we assume that the symbols belonging to a domain are known, a priori). Let I_D refer to the subset of I corresponding to the symbols S_D and let $I_{\overline{D}}$ refer to the remainder of I.

Definition A.1 *A fact f in T is said to be about domain D, or* D-dependent *if and only if there is some new interpretation N_D for the symbols S_D such that f does not hold under the interpretation $N = I_{\overline{D}} \cup N_D$.*

$$\not\models_N f$$

As an example, the fact $ImmunoSuppressed(Joe)$ is medicine-dependent since the symbol *ImmunoSuppressed* belongs to the domain of medicine, and the truth of the fact depends on the interpretation of that symbol. This fact is also Joe-dependent. In contrast, the fact $ImmunoSuppressed(Joe) \iff ImmunoSuppressed(Joe)$ is medicine-independent. Since it is a tautology, its truth does not depend on the meanings of either the symbols *ImmunoSuppressed* or Joe.

The definition of domain-dependence for meta-facts is similar to that for base level facts, but with a slight twist; symbols that *refer* to domain symbols or to domain-dependent facts are considered to be domain meta-symbols. Let I' be the intended interpretation of the set of symbols S' in a meta-theory T' of T. Let S'_D refer to the subset of symbols in S' that belong to domain D or which refer to

symbols in the domain D or to D-dependent facts in T. Let I'_D refer to the subset of I' corresponding to the symbols S'_D and let $I'_{\overline{D}}$ refer to the remainder of I'.

Definition A.2 *A fact f' in the meta-theory T' is said to be about domain D, or* D-dependent *if and only if there is some new interpretation N'_D for the domain meta-symbols S'_D such that f' does not hold under the interpretation $N' = I'_{\overline{D}} \cup N'_D$:*

$$\not\models_{N'} f' .$$

As an example, the meta-fact

$$InPremise(r, \text{"}ImmunoSuppressed\text{"}) \Longrightarrow Better(r, x)$$

is medicine-dependent since its truth depends on the meaning of the meta-symbol "*ImmunoSuppressed*" that refers to a the medical domain symbol *ImmunoSuppressed*. Likewise, the meta-fact $Better(Rule58, Rule63)$ is medicine-dependent if the symbols *Rule*58 or *Rule*68 refer to medicine-dependent facts.

These definitions are similar to definitions suggested by Greiner [GG83] for the concepts of relevance and novelty.

Bibliography

[Bar84] J. A. Barnett. How much is control knowledge worth?: a primative example. *Artificial Intelligence*, 22(1), 1984.

[BF81] A. Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence.* Volume 1, HeurisTech Press, Stanford, CA., 1981.

[Bla68] F. Black. A deductive question-answering system. In M. Minsky, editor, *Semantic Information Processing*, pages 354–402, MIT, Cambridge, 1968.

[Boy71] Robert S. Boyer. *Locking: A Restriction of Resolution.* PhD thesis, University of Texas at Austin, August 1971.

[BS84] B. G. Buchanan and E. H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, Mass., 1984.

[Cha80] C. L. Chang. On evaluation of queries containing derived relations in a relational database. In H. Gallaire, J. Minker, and J. M. Nicolas, editors, *Advances in Data Base Theory*, pages 235–260, Plenum Press, New York, 1980.

[CL81] W. J. Clancey and R. Letsinger. *NEOMYCIN: Reconfiguring a Rule-based Expert System for Application to Teaching.* Knowledge Systems Laboratory Report KSL-81-2, Stanford University, February 1981.

[Cla83] W. J. Clancey. The epistemology of a rule-based expert system. *Artificial Intelligence*, 20(3):215–251, 1983.

[Cla84] W. J. Clancey. Classification problem solving. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 49–55, American Association for Artificial Intelligence, Austin,TX, August 1984.

[Dav80] Randall Davis. Meta-rules: reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.

[dDSS79] J. deKleer, J. Doyle, G. L. Steele, and G. J. Sussman. Explicit control of reasoning. In P. H. Winston and R. H. Brown, editors, *Artificial Intelligence: An MIT Perspective*, pages 93–116, MIT Press, Cambridge, Ma., 1979.

[Die84] Tom Dietterich. *Constraint Propagation Techniques for Theory-Driven Data Interpretation.* Technical Report STAN-CS-84-1030, Stanford University, December 1984.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Doy80] J. Doyle. *A Model for Deliberation, Action, and Introspection.* Artificial Intelligence Laboratory Memo AI-TR-581, Massachusetts Institute of Technology, May 1980.

[FGKM80] E. A. Feigenbaum, Michael R. Genesereth, S. J. Kaplan, and D. J. Mostow. Intelligent agents. 1980. Proposal to the Defence Advanced Research Projects Agency.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics*, pages 19–32, American Mathematical Society, Providence, R.I., 1967.

[Gar73] M. R. Garey. Optimal task sequencing with precedence constraints. *Discrete Math.*, 4:37–56, 1973.

[Gel63] H. Gelernter. Realization of a geometry-theorem proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 134–152, McGraw-Hill, New York, 1963.

[Gen83] Michael R. Genesereth. *The MRS Casebook.* Knowledge Systems Laboratory Report KSL-83-26, Stanford University, June 1983.

[Gen85a] Michael R. Genesereth. *The* LOCK *Manual.* Knowledge Systems Laboratory Report, Stanford University, April 1985.

[Gen85b] Michael R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1985.

[GG83] R. Greiner and Michael R. Genesereth. What's new? a semantic definition of novelty. In *Proceedings of the Eigth International Joint Conference on Artificial Intelligence*, pages 450–454, Karlsruhe, W. Germany, August 1983.

[Gre69] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 183–205, American Elsevier, New York, 1969.

[GS82] Michael R. Genesereth and D. E. Smith. *Meta-Level Architecture.* Knowledge Systems Laboratory Report KSL-81-6, Stanford University, December 1982.

[GS85] Michael R. Genesereth and D. E. Smith. *Procedural Hints in the Control of Reasoning.* Knowledge Systems Laboratory Report KSL-84-11, Stanford University, January 1985.

[Hay73] P. J. Hayes. Computation and deduction. In *Proceedings of the Symposium on Mathematical Foundations of Computer Science*, pages 105–117, Czechoslovakian Acadamy of Sciences, 1973.

[Hay77] P. J. Hayes. In defence of logic. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 559–565, Cambridge, MA, August 1977.

[Hay84a] Patrick J. Hayes. Naive physic I:ontology for liquids. In Jerry Hobbs, editor, *Formal Theories of the Commonsense World*, pages 71–107, Ablex, Hillsdale, N.J., 1984.

[Hay84b] Patrick J. Hayes. The second naive physics manifesto. In Jerry Hobbs, editor, *Formal Theories of the Commonsense World*, pages 1–36, Ablex, Hillsdale, N.J., 1984.

[Hay85] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, 1985.

[HN84] L.J. Henschen and S.A. Naqvi. Compiling queries in recursive first order databases. *Journal of the ACM*, 31(1):47–85, January 1984.

[Kin81] Jonathan King. *Query Optimization by Semantic Reasoning.* Technical Report STAN-CS-81-857, Stanford University, May 1981.

[Kla78] Philip Klahr. Planning techniques for rule selection in deductive question-answering. In D. Waterman and Hayes-Roth R., editors,

Pattern-Directed Inference Systems, pages 223–239, Academic Press, New York, 1978.

[Kow70] R. Kowalski. *Studies in the Completeness and Efficiency of Theorem Proving by Resolution.* PhD thesis, University of Edinburgh, 1970.

[Kow75] R. Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22(4):572–595, October 1975.

[Kow79] R. Kowalski. *Logic for Problem Solving.* North Holland, New York, 1979.

[Kra84] Glenn Kramer. December 1984. personal communication.

[Kun84] J. Kunz. *Use of Artificial Intelligence and Simple Mathematics to Analyze a Physiological Model.* PhD thesis, Stanford University, 1984.

[Lai83] J. E. Laird. *Universal Subgoaling.* PhD thesis, Carnegie-Mellon University, 1983.

[LBAL80] R. Lindsay, B. G. Buchanan, Feigenbaum E. A., and J. Lederberg. *DENDRAL.* McGraw-Hill, New York, 1980.

[Len84] D. Lenat. Computer software for intelligent systems. *Scientific American*, 251(3):204–213, September 1984.

[Lew75] H. R. Lewis. *Cycles of Unifiability and Decidability by Resolution.* Technical Report, Aiken Computation Laboratory, Harvard University, 1975.

[LR81] D.W. Loveland and C.R. Reddy. Deleting repeated goals in the problem reduction format. *Journal of the ACM*, 28(4):646–661, October 1981.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw Hill, New York, 1974.

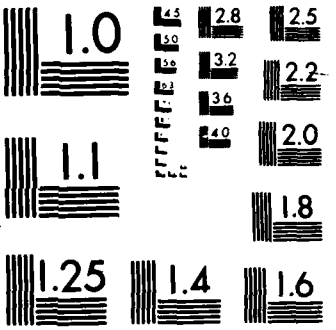
[McC68] J. McCarthy. Programs with commmon sense. In M. Minsky, editor, *Semantic Information Processing*, pages 403–418, MIT, Cambridge, 1968.

[McD77] D. V. McDermott. *Flexibility and Efficiency in a Computer Program for Designing Circuits.* Artificial Intelligence Laboratory Memo AI-TR-402, Massachusetts Institute of Technology, 1977.

[McD80] J. McDermott. R1: an expert in the computer systems domain. In *Proceedings of the First National Conference on Artificial Intelligence*, pages 269–271, American Association for Artificial Intelligence, Stanford,CA, August 1980.

AD-A172 502 CONTROLLING INFERENCE(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE D E SMITH APR 86 STAN-CS-86-1107 N00014-81-K-0004 3/3

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0

[McD84] Drew V. McDermott. Reasoning about plans. In Jerry Hobbs, editor, *Formal Theories of the Commonsense World*, pages 269–317, Ablex, Hillsdale, N.J., 1984.

[MG85] J. Mackinlay and Michael R. Genesereth. Expressiveness and language choice. *Data & Knowledge Engineering*, 1(1):17–29, 1985.

[MN83] J. Minker and J. Nicolas. On recursive axioms in deductive databases. *Information Systems*, 8(1):1–13, 1983.

[Moo75] R. C. Moore. *Reasoning From Incomplete Knowledge in a Procedural Deduction System.* Artificial Intelligence Laboratory Memo AI-TR-347, Massachusetts Institute of Technology, December 1975.

[MS81] D. P. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 368–374, Vancouver, August 1981.

[MW77] Z. Manna and R. Waldinger. *Studies in Automatic Programming Logic.* North-Holland, New York, 1977.

[Nau86] Jeff Naughton. Data independent recursion in deductive databases. In *Proceedings of the ACM/SIGMOD Symposium on Principles of Database Systems*, Association for Computing Machinery, 1986.

[NH80] S. A. Naqvi and L. J. Henschen. Performing inferences over recursive data bases. In *Proceedings of the First National Conference on Artificial Intelligence*, pages 263–265, American Association for Artificial Intelligence, Stanford, CA, August 1980.

[Nil80] N. J. Nilsson. *Principle of Artificial Intelligence.* Tioga, Palo Alto, 1980.

[NSS63] A. Newell, J. Shaw, and H. Simon. Empirical explorations with the logic theory machine: a case study in heuristics. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 109–133, McGraw-Hill, New York, 1963.

[Pea84] J. Pearl. *Heuristics.* Addison-Wesley, Reading, Mass., 1984.

[PP82] L. M. Pereira and A. Porto. Selective backtracking. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 107–114, Academic Press, New York, 1982.

[Rai70] H Raiffa. *Decision Analysis: Introductory Lectures on Choices under Uncertainty.* Addison-Wesley, Reading, Mass., 1970.

[Rei78a] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76, Plenum Press, New York, 1978.

[Rei78b] R. Reiter. On structuring a first order data base. In Ray Perrault, editor, *Proceedings of the Second National Conference*, pages 19–21, Canadian Soc. Computational Studies of Intelligence, Toronto, July 1978.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1):81–132, 1980.

[Row83] Neil Rowe. *Rule-based Statistical Calculations on a Database Abstract.* Technical Report STAN-CS-83-975, Stanford University, June 1983.

[RS83] J. Rosenschein and V. Singh. *The Utility of Meta-level Effort.* Knowledge Systems Laboratory Report KSL-83-20, Stanford University, March 1983.

[Rus85] Stuart Russell. *The Compleat Guide to* MRS. Technical Report STAN-CS-85-1080, Stanford University, June 1985.

[SAC*79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data*, pages 23–34, Association for Computing Machinery, 1979.

[SG85] David E. Smith and Michael R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26(2):171–215, 1985.

[Sho84] E. H. Shortliffe. Details of the consultation system. In B. G. Buchanan and E. H. Shortliffe, editors, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, pages 78–132, Addison-Wesley, Reading, Mass., 1984.

[Sin85] Narinder Singh. *Exploiting Design Morphology to Manage Complexity.* PhD thesis, Stanford University, August 1985.

[SK75] Herbert Simon and J. Kadane. Optimal problem-solving search: all-or-none solutions. *Artificial Intelligence*, 6:235–247, 1975.

[Smi83] D. E. Smith. Finding all of the solutions to a problem. In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 373–377, American Association for Artificial Intelligence, Washington D.C., August 1983.

[Spr77] Robert Sproull. *Strategy Construction Using a Synthesis of Heuristic and Decision-theoretic Methods.* PhD thesis, Stanford University, 1977.

[SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[Sti82] Mark Stickel. A nonclausal connection-graph resolution theorem proving program. In *Proceedings of the Second National Conference on Artificial Intelligence*, pages 229–233, American Association for Artificial Intelligence, Pittsburgh, August 1982.

[Tre86] R.J. Treitel. *Sequentialising Logic Programs.* PhD thesis, Stanford University, 1986. In preparation.

[Ull82] Jeffrey D. Ullman. *Principles of DataBase Systems.* Computer Science Press, Rockville Maryland, 1982.

[Ull84] Jeffrey D. Ullman. *Implementation of Logical Query Languages for Databases.* Technical Report STAN-CS-84-1000, Stanford University, May 1984.

[Ull85] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(4), September 1985.

[War81] David Warren. Efficient processing of interactive relational database queries expressed in logic. In *Proceedings of the seventh VLDB conference*, pages 272–281, IEEE, 1981.

[WY76] E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241, September 1976.

END

11-86

DTIC